Computer Science Department

TECHNICAL REPORT

A Single-Pass Algorithm for Eliminating Constraint Checks in Ada Programs

H. Operowsky
E. Schonberg

Technical Report 430

December 1988

NEW YORK UNIVERSITY

Department of Computer Science
Courant Institute of Mathematical Sciences
251 MERCER STREET, NEW YORK, N.Y. 10012

A Single-Pass Algorithm for Eliminating Constraint Checks in Ada Programs

H. Operowsky
E. Schonberg

Technical Report 430

December 1988

A Single-Pass Algorithm for Eliminating Constraint Checks in Ada Programs

Howard L. Operowsky
IBM Corporation
Data Systems Division
36 Apple Ridge Road
Danbury, CT 06810

Edmond Schonberg
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University
251 Mercer Street
New York, NY 10012

December 26, 1988

Abstract

We show that a global optimizer is not required to reduce the overhead of constraint checking in Ada programs. We present a series of data-flow equations for available expressions and use them as the basis for a simple algorithm to eliminate redundant constraint checks. The algorithm is syntax-directed and is executed in a single pass over the source program's abstract syntax tree. No control flow analysis is required. Our algorithm also includes constant propagation using an extended framework and induction variable analysis. Because the algorithm operates on the abstract syntax tree, induction variable analysis is simplified. Although programs with **goto** statements are not considered, the **exit** statement is handled fully. We also examine the effects of shared variables and exception handling.

1 Introduction

In order to enhance program reliability, Ada provides six predefined exceptions to handle abnormal run-time conditions. These exceptions are: *constraint_error, numeric_error, program_error, storage_error, and tasking_error.* The most general of these is the constraint_error, which the Ada Language Reference Manual (LRM) [DoD83] specifies be raised in the following situations:

> ...upon an attempt to violate a range constraint, an index constraint, or a discriminant constraint; upon an attempt to use a record component that does not exist for the current discriminant values; and upon an attempt to use a selected component, an indexed component, a slice or an attribute, of an object designated by an access value, if the object does not exist because the access value is null.

The generated code for a range or index check consists of a test that the expression is greater than or equal to the lower bound of the indicated range, and less than or equal to the upper bound of that range. The generated code for a discriminant check or access check is a test of equality. Thus, all constraint checks may be generalized to assume the following form:

$$check\_type(expression, constraint)$$

where *check_type* is equal(eq), greater than or equal(ge), or less than or equal(le), and *constraint* is the lower or upper bound of a range, the discriminant of a record object or constrained record subtype, or some constant value. For example, the check that the value of *i* is within the range for the subtype *some* would be specified by the pair of tests

```
ge(i, some'First)   -- if i < some'First then raise constraint_error;
le(i, some'Last)    -- if i > some'Last then raise constraint_error;
```

In the remainder of the chapter, *test* and *check* will be used as synonyms for the term constraint check.

Unfortunately, because these checks are performed at run-time, there is a (possibly severe) performance penalty associated with them. In recognition of that penalty, the language provides the programmer with the means of not generating these tests through use of the **SUPPRESS pragma**.

This paper presents an algorithm to eliminate redundant constraint checks. The algorithm differs from the usual algorithms found in optimization literature in that it does not require a preliminary pass for control flow analysis. Nor does it compute the def-use chains normally required for data flow analysis. Instead, it is driven by the syntax of the program itself. Such syntax-directed optimization has been pioneered by Geschke [Ges73] and implemented in an early Bliss compiler [WJW+75]. Powell [Pow84] also reports using a similar technique for a Modula-2 compiler, based in part on the notion of *linear regions* described by Cocke and Schwartz [CS70].

The advantage of the algorithm presented here is its simplicity. It is presented as a single pass over the syntax tree. No complicated data structures are required to maintain the information required for the analysis. Thus, a considerable amount of constraint checking may be eliminated at compile time without requiring a general purpose optimizer. To do so, the algorithm depends on the good programming habits that Ada is supposed to instill. Strong typing will tend to result in multiple references to variables of the same type, based on the same type template. Thus, a simple routine capable of recognizing available expressions can be quite effective. We also depend on the programmer's avoidance of the **goto** statement. Programs with explicit **gotos** are not considered. We do consider other escape mechanisms within the language. The **exit** statement is handled fully by the algorithm, and we describe the extensions required to support Ada's exception handling facility.

The algorithm is composed of three main components: *available expression elimination, constant propagation*, and *induction variable analysis*.

Available Expression Analysis. Available expression analysis is clearly the focus of the algorithm as it attempts to eliminate repetitious constraint checks. This phase depends heavily on the assumption that most references within a statement will be to objects which are described by the same template and will naturally cause redundant constraint checks to be generated by a naive compiler.

Constant Propagation. The second algorithm applied is constant propagation. Since the algorithm is applied over a single pass of the tree, without benefit of the def-use chains typically used in these algorithms, it is somewhat rudimentary. However, the goal of the algorithm is

to eliminate obviously unnecessary tests with minimal expense. One extension has been made to the typical constant propagation framework. An additional level has been added to the standard lattice to account for the sign of the variable. This is important because many common constraint checks test whether an expression lies within some positive range, rather than for equality to some quantity. Thus, if we can determine the sign of some expression, we may be able to eliminate the check on one of the end points of the range even when we don't know the values of the variables themselves. Common use of the predefined subtypes `Natural` and `Positive` make this potentially useful.

Induction Variable Analysis. Finally, the algorithm performs induction variable analysis. The purpose of this analysis is to replace tests found in the body of a loop and which reference induction variables with equivalent tests in the prologue of the loop. Ada is more likely to have auxiliary induction variables than other procedural languages because the **for** loop allows only unit steps. Furthermore, the range of the loop parameter is clearly defined in the corresponding syntax tree. Thus, the entire range of values assumed in the loop by any induction variable may easily be computed. Such an optimization would be much more difficult if performed on a lower-level intermediate representation such as quadruples or triples in which the limits of the loop parameter are obscured.

Section 2 contains a more detailed description of the algorithm. As we indicated earlier, programs with explicit **goto** statements are not considered, but **exit**s are handled fully. We also restrict ourselves to single-task programs which do not contain programmer-written exception handlers. We then discuss the extensions to the algorithm for tasking and exception handling in Sections 3 and 4 respectively. Section 5 illustrates the effects of the algorithm on several programs. Section 6 presents the data structures used by the algorithm discussed in Section 2, and Section 7 presents a pseudo-code version of that algorithm. Finally, Section 8 presents a proof of the correctness of the algorithm.

2 Description of the algorithm

2.1 Available Expression Elimination

Available expression elimination is an example of a data flow problem characterized by forward flow with a meet operation of intersection. This section describes the data flow rules used by the algorithm to eliminate redundant constraint checks. In Figures 1–6, the S enclosed in the box on the left represents some statement which is expanded into the sequence of statements drawn to its right. The data flow equations for the construct on the right are presented immediately below each diagram. These formulas reference the following sets:

gen(S) = the set of constraint checks required by statement S.

kill(S) = the set of constraint checks, created outside of S, but which are invalidated by side effects of statement S.

in(S) = the set of constraint checks already inserted into the program and still valid at the point immediately before S.

out(S) = the set of constraint checks valid at the point immediately after statement S.

In Figure 1, a *simple statement* refers to a statement which cannot itself contain other statements. A sequence of simple, conditional, loop, and/or **exit** statements is called a *linear block* throughout the remainder of this document, and is the basic control structure processed by the algorithm. The term "linear block" corresponds to the term *sequence_of_statements* used in the LRM, but is used here instead to emphasize the strict linear order in which the algorithm processes such sequences. A linear block differs from a *basic block* in standard optimization terminology in that a linear block does not contain purely sequential control flow *within* it. The algorithm processes each linear block in a sequential order by invoking itself whenever it reaches a compound statement, and processing each linear block of the compound statement in the manner defined by the previous formulas.

The algorithm begins with the first statement of the outermost linear block in the procedure. Given the formulas above, the general flow of the algorithm may be described as follows.

2.1.1 Simple Statement

In simple terms, the algorithm for processing simple statements is:

1. Scan the statement and construct the list of necessary tests (*gen(S)*).
2. Compare each test on the list with the set of tests currently alive and eliminate any redundant ones.
3. Add the remaining tests to the uselists for each variable referenced in the test.
4. Kill all tests invalidated by side effects of the statement.
5. Go to the next statement.

Now let's consider these steps in more detail.

2.1.1.1 Scan the statement. Each statement is examined and all required template tests are exposed. Range and index tests are split into separate tests for the upper and lower bounds. In addition, this phase attempts to eliminate tests on the basis of the information gathered during constant propagation. If a range check is required, the ranges of the referenced objects are first checked. If the ranges of all referenced objects, as well as the constraint, are compile time constants, then this phase determines whether a run-time check is necessary by constant folding the bounds of the component ranges into the expression. If any of the component variables has a constant value, regardless of whether its range is constant, then that value is of course used.

The form of the statement is inspected. If the statement is of the form

$$x := x + (some\ expression);$$

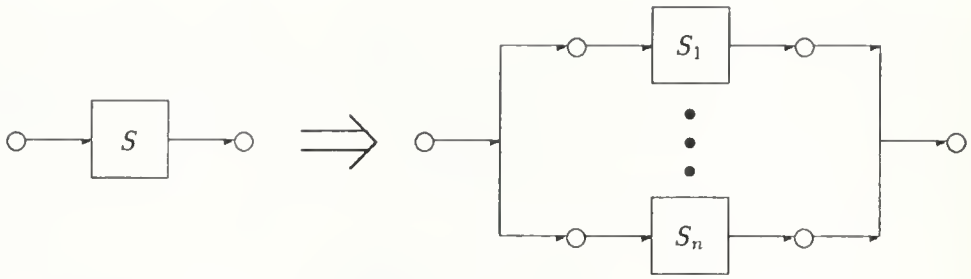
constant propagation is to be used to determine the sign of the expression. If the sign can be determined, then only one test might be generated.

$$
\begin{aligned}
gen(S) &= gen(S_1) \\
kill(S) &= kill(S_1) \\
in(S_1) &= in(S) \\
out(S) &= gen(S) \cup \Big(in(S) - kill(S)\Big)
\end{aligned}
$$

Figure 1: Simple Statement

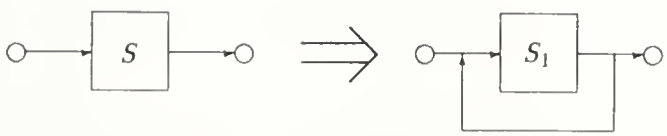
$$
\begin{aligned}
gen(S) &= gen(S_2) \cup \Big(gen(S_1) - kill(S_2)\Big) \\
kill(S) &= kill(S_2) \cup \Big(kill(S_1) - gen(S_2)\Big) \\
in(S_1) &= in(S) \\
in(S_2) &= out(S_1) \\
out(S) &= out(S_2)
\end{aligned}
$$

Figure 2: Sequence of Statements



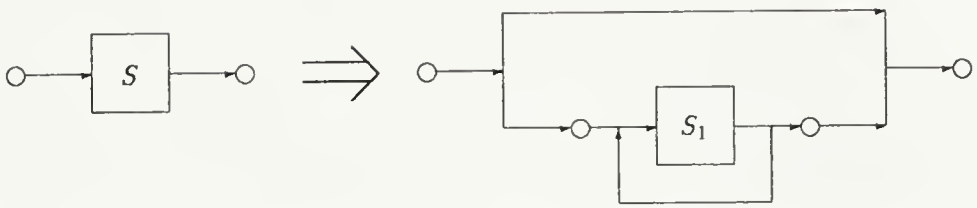
$$
\begin{aligned}
gen(S) &= \bigcap_{i=1}^{n} gen(S_i) \\
kill(S) &= \bigcup_{i=1}^{n} kill(S_i) \\
in(S_1) &= \cdots = in(S_n) = in(S) \\
out(S) &= \bigcap_{i=1}^{n} out(S_i)
\end{aligned}
$$

Figure 3: Conditional Statement



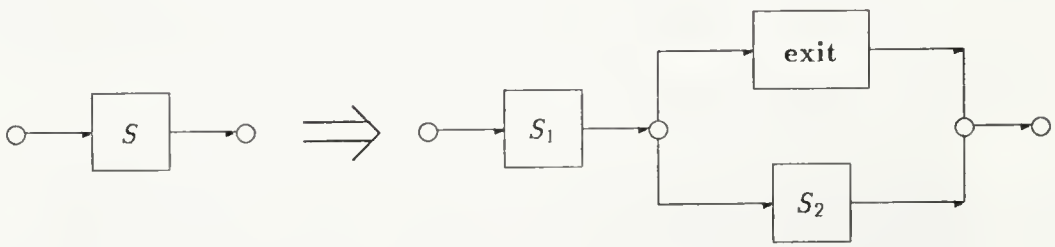
$$
\begin{aligned}
gen(S) &= gen(S_1) \\
kill(S) &= kill(S_1) \\
in(S_1) &= in(S) \cap out(S_1) \\
&= in(S) - kill(S_1) \\
out(S) &= out(S_1) \\
&= gen(S_1) \cup \Big(in(S) - kill(S_1)\Big)
\end{aligned}
$$

Figure 4: One-Trip Loop



$$
\begin{aligned}
gen(S) &= gen(S_1) \cap gen(\emptyset) \\
&= \emptyset \\
kill(S) &= kill(S_1) \\
in(S_1) &= in(S) - kill(S_1) \\
out(S) &= out(S_1) \cap in(S) \\
&= \Big(gen(S_1) \cup \big(in(S_1) - kill(S_1)\big)\Big) \bigcap in(S) \\
&= \Big(gen(S_1) \cup \big(in(S) - kill(S_1)\big)\Big) \bigcap in(S) \\
&= in(S) - kill(S_1)
\end{aligned}
$$

Figure 5: Zero-Trip Loop



$$
\begin{aligned}
gen(S) &= gen(S_1) - kill(S_2) \\
kill(S) &= kill(S_1) \cup kill(S_2) \\
in(S_1) &= in(S) \\
in(S_2) &= out(S_1) \\
out(S) &= out(S_1) - kill(S_2)
\end{aligned}
$$

Figure 6: Exit Statement

Care must be taken in deciding how to represent the constraint that is being checked. Tests are created on the "most general" object possible. For example, if an array is of a subtype known at compile time, then subscript range checks are based on the *subtype* of the array rather than the array itself. Thus, it may be possible to eliminate later tests on other arrays of the same subtype. Similarly, if the discriminant of a record is used as one of the bounds of an array component, any subscript range check references the discriminant rather than the array bound that the discriminant defines. For example, in the following code fragment

```
1.  type variable_string(length : Integer) is
2.     record
3.        data : String(1..length);
4.     end record;
5.  varstr : variable_string(len);
```

range checks on *varstr.data* reference *varstr.length* rather than *varstr.data*'Last. The advantage of this is that the algorithm can use the range checks to also eliminate any later tests which reference the discriminant of *varstr* explicitly. In general, by increasing the number of tests in which a name appears, we increase the likelihood that later tests will be found to be redundant.

2.1.1.2 Compare against currently valid tests. After inspection of the statement, each test required by the language is compared to all previous checks which have been generated against the same constraint, and which are still valid. For example, an index check of the form *le(I, A*'Last*)* (i.e., I $\leq$ A'Last) is required for the statement statement $A(I) := A(I) + 1$;. But before actually inserting this test into the syntax tree, the list of currently valid tests of the form $*(*, A$'Last) is scanned in search of $le(I, A$'Last). A test is valid if both

1. none of the operands of the expression being checked has been changed, and

2. it dominates the statement currently being processed.

Under those conditions, there is no benefit in executing the test again. If the expression violates the test, the error would already have been exposed by the earlier check. In many cases, a check will be redundant due to a test created in the current linear block, or in some outer block containing the current linear block. However, the algorithm also recognizes constraint checks which appear on all branches of a conditional statement, and checks generated within 1-trip loops[1].

A constraint check is killed whenever the expression being validated is killed through an assignment to any of the variables referenced in the expression. Changes to a variable are indicated by a value known as its *store number*. Store numbers are a monotonically increasing sequence of numbers. When a variable is modified, the current value of the store number counter is stored into the variable's symbol table entry, and the counter is incremented. When a constraint check is created, the current store number for each referenced variable is stored into the tree node for that variable in the expression. When a new linear block is entered, the current value of the store number counter is stacked to identify variables defined within the

[1] A *1-trip loop* is a loop which the compiler can determine, at compile-time, will be executed at least once.

linear block. The need for such a stack is explained later in the sections dealing with conditional statements and loops.

As the name implies, store numbers are simply meant to discriminate between successive values of each single variable. They are simpler than *value numbers*[CS70]; they are never shared, nor are they ever applied to expressions. They are weaker than value numbers in that, since they are never shared, they prevent the algorithm from recognizing and eliminating redundant tests of different expressions with equal values. Value numbers, on the other hand, incur considerably more overhead; tables must be maintained to provide the value number of any expression, as well as describe the set of expressions associated with every value number. Given the existence of the syntax tree, it seems just as efficient to compare the subtrees of the expressions being tested as to hash each of the expressions to find their value numbers. If, however, the increased number of tests eliminated by value numbers warrants the increased processing cost, the store numbers could easily be replaced. The basic algorithm is unchanged regardless of the technique used to identify modified variables.

Checks are then identical if the expressions are structurally identical and the store numbers for each variable are equal. (As indicated later in the section on data structures, the tests also contain the highest value number in the expression to quickly eliminate tests from consideration.) In addition, bounds checks are eliminated if there is another live test which covers a larger sequence of values. This is particularly useful because of the simple expressions commonly used as array subscripts. Consider, for example, the statement $x(i) := x(i+1);$. The compiler must ensure that both i and $i+1$ are within x'`Range`, but it needn't compare both expressions to both endpoints of the range to prove that. The test $le(i+1, x\text{'}\texttt{Last})$ certainly obviates the test $le(i, x\text{'}\texttt{Last})$. Any index or range check testing an expression of the form *variable* $\pm$ *expr*, where the sign of *expr* is known, is flagged as "simple." (In practice, it is probably sufficient that *expr* be a constant.) If the new test and the existing test are both "simple" then instead of direct comparison, the search algorithm determines if the expression in the new test is within the range tested by the existing expression. If so, then the new test can be discarded. If the new test covers a superset of the range tested by the existing set, and both tests are within the same statement, then the new test replaces the existing test. Because of the nature of array subscripts, this optimization should yield significant results.

2.1.1.3 Add New Tests to the Program. If no equivalent test is available, then the test is added to a list of tests attached to the syntax tree node for the statement. In addition, the test is added to a *uselist* for each variable referenced in the test, as well as the template field or constant that is being tested. If the object is of a constrained subtype, then it is expected that there will be an entry for the subtype (and hence the template) in the symbol table, and that this entry will act as anchor for the list of associated tests. On the other hand, if the object is of an unconstrained subtype, then it is impossible to associate a template with the object, so the uselist for the object is used instead. Formal parameters are also handled in this manner.

2.1.1.4 Account for side-effects of the statement. After the tests have been processed, the side-effects of the statement are accounted for. Every variable modified by the statement is given a new store number, and each test on the variable's uselist is invalidated to prevent later tests from being erroneously eliminated. If the variable is a composite object, then the

standard conservative approach is taken. An assignment to any element, $a(i)$, of an array a causes all tests which reference any element of the array to be invalidated. Similarly, an assignment to any component, $r.c$, of a record r causes all tests which reference any component of the record to be invalidated. In general, tests which reference *attributes* of the composite object, such as `'First` of some dimension of an array, as opposed to a value, such as $a(2)$, need not invalidated. If, however, the object is of an unconstrained record or access subtype, then it is possible that an assignment to the object can change its attributes as well. For example, if r is an object of an unconstrained record subtype with an array component $r.a$, the shape of $r.a$ may be changed by assigning an aggregate to r. Currently, the algorithm does not discriminate between assignments to composite objects. All associated tests, including those which only reference attributes, are killed on every assignment to a composite object. Similarly, every definition of a designated object, $ao.d$, invalidates all tests which reference designated objects of the same type as $ao.d$. Processing then continues with the next statement in the sequence.

If the next statement is a simple statement, then the processing repeats exactly as above. If the next statement is a compound statement, then a recursive call of the algorithm processes the linear blocks contained within it, and returns lists of constraint checks generated and external checks killed by the statement. The following sections on conditional and loop statements describe the additional actions taken to account for the control flow indicated by these compound statements.

2.1.2 Conditional Statements

To facilitate the discussion, each of the alternate sequences of statements in the conditional statement is termed a *conditional block*.

In most global optimization algorithms, control flow analysis is used to determine possible control flow through the program and to create a graph of that flow. Data flow analysis is then performed to propagate "local" information initially associated with each node or arc in the graph, to all other nodes or arcs within the flow graph that might be affected when the program is actually executed. Optimization transformations finally apply the collected information to improve the quality of the generated code. The key point to note is that information is replicated among the nodes and arcs in the control flow graph.

In the algorithm presented in this paper, each statement is processed sequentially. There is no control analysis and minimal data flow analysis prior to the code improving transformations, and there is a single, common, source of data used at each point in the program. The difficulty in processing conditional statements sequentially is that, as the data flow equations in Section 2.1 state, the *in* set for each conditional block must be the same. The same external tests must be available to each conditional block in the statement; each variable must contain the same store number upon entry into each conditional block. We shall generally refer to the *in* set together with the store number for each variable at any point in the user program as the *state* of the program at that point.

As each conditional block is processed, the state of the program changes; tests are added and killed, and variables are assigned new store numbers. In more standard optimization algorithms, such changes are made locally to the *gen* and *kill* sets associated with the statement's basic blocks, and propagated to other basic blocks as indicated by control flow analysis. The statement's other conditional blocks are unaffected by the changes (in fact, there is typically no

awareness that the conditional blocks are part of the same statement). But we do not maintain separate collections of information for each block. All changes are made to the common data source mentioned earlier. Nevertheless, we must simulate the actions of the data flow equations in Section 2.1. We must ensure that each conditional block is processed with the same initial *in* set of valid tests and the same value for every variable. In the algorithm below, this is accomplished via a recursive call for each of the conditional blocks. Upon return from each block, the procedure provides lists of all tests generated within the conditional block that are live upon exit, all *external* tests killed within the conditional block, and all variables which were redefined in the conditional block. These lists are then used to "undo" all the changes before processing the next conditional block. The newly generated tests are hidden, the newly killed tests are revalidated, and the changed variables receive the store numbers they had upon entry to the conditional statement. But all such changes are remembered to correctly compute the *out* set after all conditional blocks are processed.

Recall that the set of tests available after a conditional statement S, with conditional blocks $S_i, i = 1, \ldots, n$ is

$$out(S) = \bigcap_{i=1}^{n} out(S_i)$$

To compute the intersection of tests generated in the conditional blocks, the list of tests generated in each conditional block is compared against tests generated in previous conditional blocks. Congruent tests (tests which are syntactically identical, but which do not necessarily have the same value) appearing in every conditional block are made available to the remainder of the program.

Conversely, all killed tests which were revalidated are killed again (permanently), and any variable changed within the conditional statement is given a new store number.

2.1.2.1 Pseudo-Tests. It should be noted that the requirements on computing the intersection of all tests generated within branches of the conditional statement is slightly different from determining whether a test is redundant. In the latter case, not only must the expressions be the same, but each of the operands must be identical in value. In computing the intersection, however, it is only necessary that the expressions be identical. The actual value of the expression is unimportant. If such tests are found, a new "pseudo" test is generated which reflects the value of each component variable upon exit from the conditional statement. Pseudo tests are ignored by the code generator. Their sole purpose is to reflect a condition that has been validated in some other manner.

Pseudo tests are also created to reflect constraint checks which are coded by the programmer. Consider the code fragment

if *data.record_type* = *employee* **and then** *data.name* = "JONES" **then**

where *name* is a variant component of a record type with discriminant *record_type*. In general, the compiler would be required to generate a test on the value of *data.record_type* before it could access the *name* field. In this case, the programmer has checked this for himself and so no test is necessary. Pseudo tests provide a simple means of making such information available to the algorithm.

The above fragment also exemplifies the use of the short-circuit control forms **and then** and **or else** in coding a boolean expression which could result in an error if **and** or **or** were used. Interestingly, these special control forms add to the difficulty of determining the visibility of tests generated for the boolean expression guarding each conditional block. Once the result of the short-circuit expression may be determined, the remainder of the tests are bypassed. Thus, it is impossible to tell in advance exactly which tests (except for the first) have been executed. Similarly, the short-circuit control forms impede creation of pseudo tests from test conditions. In the algorithm below, it is assumed that if short-circuit control forms are used in a boolean expression, only the first constraint check is live outside the expression itself.

No attempt has been made to propagate failed conditions to succeeding conditional parts of the statement. For example, if the statement is of the form:

if c_1 **then** s_1 **else** s_2;

no pseudo tests corresponding to (**not** c_1) are available to s_2. Such an extension could be added at a later time if warranted (such an algorithm appears in [AKPW83]). However, it is quite likely that the constraint checks within c_1, which will be available to s_2, will prove to be more useful in eliminating checks in s_2, regardless of the value of c_1.

2.1.3 Loops

The concern in processing loops in a single forward pass over the program is that existing external tests, apparently valid at the top of the loop, may in fact be killed by code later in the loop. This is typically handled during data flow analysis. Our algorithm handles this by making a single pass over the loop, and compiling a list of all variables defined in the loop. This list is appended to the tree node representing the loop header. If the loop contains another loop nested within it, that loop is processed sequentially as well. The list of variables defined within the nested loop is attached to the loop node for the nested loop and added to the list of defined variables for the outer loop. Before processing the body of any loop, the current value of the store number counter is stacked, and each variable in the list is assigned a new store number, thus immediately invalidating all external tests which will eventually be killed within the loop. Loop invariant expressions are recognized by means of the store number on the top of the store number stack. If the store number for each operand of an expression is less than the store number on the top of the stack, then the expression must be loop invariant. Nested loops are processed in a similar manner. Once all external tests are invalidated, the body of the loop may be processed in a straightforward manner. After the body has been processed, loop invariant constraint checks generated within the loop are moved to the prologue of the loop.

As the data flow equations above indicate, the set of tests available upon exit from a loop is different depending on whether or not the loop is unconditionally executed at least once. Loops whose entry condition guarantees that their bodies will be executed at least once are called *1-trip loops*; loops whose bodies might not be executed at all are called *0-trip loops*. Because 0-trip loops might never be executed, tests generated by them may not be visible to code following the loop. Thus, after the linear block associated with the loop has been processed, all tests generated within the block must be killed. External tests killed by 0-trip loops remain invalid for safety reasons.

As in the case of conditional statements, the boolean expression guarding entry into the loop is examined and pseudo tests are created, when appropriate, prior to processing the body of the loop.

2.1.4 Exit Statements

Another problem associated with loops is the processing of **exit** statements. A key simplifying assumption of the syntax-directed approach employed in this algorithm is that each statement has a single entry and a single exit. The **exit** statement clearly violates that assumption. Although a well-behaved variant of the **goto** statement, the **exit** statement is difficult to process in this framework because, unlike the conditional statement or (unexited) loop, it is capable of transferring control to a point multiple levels away. Indeed, loops may have multiple exits to various enclosing loops. Pictorially, these exits could produce control flow graphs with intersecting arcs. The present algorithm is efficient precisely because it is unnecessary to construct a control flow graph. Unlike analysis on a lower-level representation, there is no preliminary pass to reconstruct control flow. Fortunately, use of the **exit** statement is relatively rare; multiple level exits are rarer still.

Because **exits** in Ada may be conditionally executed, it is unsafe to assume that any tests constructed after the **exit** statement are available outside the sequence of statements in the current linear block following the **exit** statement. Thus, the algorithm treats those statements as a separate conditional block and processes them separately. As with any conditional block, pseudo-tests are created as appropriate. When that block has been completely processed, all tests created within it are killed. Conversely, any external test killed within it remains invalid thereafter.

However, the algorithm must remember the outermost level to which any **exit** in the current linear block returns, so that enclosing linear blocks can be processed correctly. When control returns from a linear block, an indication of the outermost loop exited by the block is also returned. If the outermost loop is the loop currently being processed, then no further action is required other than that already described. If, instead, some enclosing loop is exited, then it is possible that none of the remaining statements in the current linear block will be executed. The net effect is that the compound statement now being processed must itself be viewed as an **exit** statement in its current linear block. Thus, the remainder of the block is treated separately and all tests generated after the compound statement must be killed.

Figures 7 and 8 illustrate two cases which present slight technical difficulties. In Figure 7, the **exit** for the loop, *ExitedLoop*, is contained within the conditional block for *Case2*. In that case, the conditional blocks following the *Case2* block must be handled as if there was no **exit** at all. But the **exit** must be remembered so that the remainder of *ExitedLoop* following the **end case** is handled correctly. Also, the outermost loop exited by any of the conditional blocks must be remembered.

In Figure 8, *OuterLoop* is exited from *InnerLoop1*, but not from *InnerLoop2*. Care must be taken to insure that the algorithm does not mistakenly treat *InnerLoop2* as an exited loop. To guard against that error, the algorithm uses an array to contain the outermost level exited by any loop level. As each loop is entered, the level exited is reset. After processing the body of a loop, the level exited is inspected and, if an outer level is exited, the array entry for the immediately enclosing loop is updated.

```
1.  ExitedLoop:
2.      loop
3.          ...
4.          case
5.              when Case1 ⇒
6.                  ...
7.              when Case2 ⇒
8.                  ...
9.                  exit when ...
10.             when others ⇒
11.                 ...
12.         end case;
13.         ...
14.     end ExitedLoop;
```

Figure 7: **exit** Contained Within a Conditional Block

```
1.  OuterLoop:
2.      loop
3.          ...
4.          InnerLoop1:
5.              loop
6.                  ...
7.                  exit OuterLoop when ...
8.              end InnerLoop1;
9.          ...
10.         InnerLoop2:
11.             loop
12.                 ...
13.             end InnerLoop2;
14.     end OuterLoop;
```

Figure 8: Outer Loop Exited by One of its Enclosed Loops

2.2 Constant Propagation Analysis

A significant number of constraint checks can be eliminated simply by using the constants that typically appear in declarations. The goal of the constant propagation algorithm presented here is to obtain additional information about object values from the code, in a single pass, concurrently with the available expression analysis described above. Because the primary purpose of the algorithm is to help eliminate unnecessary constraint checks, we attempt to propagate the sign of an object when its actual value cannot be determined. Thus, it may be possible to eliminate one of the bounds checks when a range check is required. To do so, the standard framework for constant propagation is extended by adding two additional values, denoting positive and negative values, to the lattice. The meet operation on these elements is presented in Figure 9. The algorithm is relatively straight-forward and only the handling of

top	$\wedge$	*anything*	=	anything, by definition of lattice
a	$\wedge$	a	=	a, where a is some constant
a	$\wedge$	b $(a \neq b)$	=	$sign(a)$, if $sign(a) = sign(b)$
			=	*bottom*, otherwise
$sign_1$	$\wedge$	$sign_1$	=	$sign_1$, where $sign_1 = +$ or $-$
$sign_1$	$\wedge$	$sign_2$ $(sign_1 \neq sign_2)$	=	*bottom*
a	$\wedge$	*sign*	=	$sign(a)$ if $sign = sign(a)$
bottom	$\wedge$	*anything*	=	*bottom*, by definition of lattice

Figure 9: Meet Operation for Constant Propagation

loops requires further explanation. Because it is performed within a single pass, it must handle loops very conservatively. Upon reaching a loop in the linear block, any variable which receives a new store number as described earlier, also has its lattice value reset to *bottom*. Assume now that the loop does not contain any **exit** statements.

If the loop is a 1-trip loop, then, because there are no **exit** statements, the last statement in the body of the loop is the sole predecessor and immediate dominator of the statement immediately following the loop. The final lattice value of each variable after the loop body is processed is $\leq$ its true value upon exit from the loop, and may safely be used by the remainder of the program.

A 0-trip loop is equivalent to a 1-trip loop contained within a conditional block and therefore requires a bit more work. In particular, we must account for the possibility that the loop is not executed. The true value of a variable upon exit from the loop is either its final value upon exit from the *body* of the loop, or its value upon entry to the preheader of the loop. We must therefore assign to each object modified in the loop a new lattice value equal to the meet of the object's lattice value upon entry into the loop and its final lattice value upon exit from the loop. We can express this mathematically as

$$out(S) = in(S) \wedge out(S_1)$$

where $out(S)$ is the lattice value of a variable upon exit from a loop S, with body S_1, and $in(S)$ is the variable's lattice value upon entry into the loop. This is in contrast to the formula for

the 1-trip loop, which is simply

$$out(S) = out(S_1).$$

Thus, the algorithm must be able to recall $in(S)$ for every variable defined within the loop, in order to correctly perform the meet operation after the loop is processed. Note that this means that variables defined within a 0-trip loop receive new store numbers, corresponding to the new lattice values, upon exit. Fortunately, available expression analysis is not affected by these changes because, as we discussed earlier, all tests generated within a 0-trip loop are killed upon exit.

2.2.1 Exit Statements

As before, the **exit** statement is more difficult to process because it violates the strict LIFO behavior of the conditional and loop statements. If the **exit** statement transfers control out of the immediate loop only, then it can be handled in a straightforward manner. A variable's lattice value upon exiting the loop is the meet of its lattice value just before the **exit** statement and its final lattice value in the body of the loop; in the case of 0-trip loops, the variable's lattice value upon entry into the loop is included in the meet operation as well. Unfortunately, the **exit** statement can transfer control to any enclosing linear block. Moreover, there may be multiple **exits** within the loop, each transferring control to a different level. In the case of test elimination, it was sufficient to simply kill all tests generated after an **exit** statement. In the case of constant propagation, we can be a bit less conservative.

Consider a block with two **exit** statements contained within it. As indicated above, when the first **exit** statement is reached, the algorithm calls itself recursively to process the remaining statements in the linear block separately. In doing so, the second **exit** statement is eventually reached, which causes a second recursive call. When the end of the linear block is reached, the lattice value of each variable modified after the second **exit** statement is modified to reflect the meet of its final lattice value in the linear block and its lattice value prior to the second **exit**. Control is then returned to the caller at the first **exit** statement and once again the lattice value of each variable modified after the **exit** is set to the meet of its lattice value upon return from the recursive call and its lattice value just prior to the **exit** statement. Upon exit from the loop, then, the lattice value of each variable modified within the loop is $\leq$ its true value.

2.3 Induction Variable Analysis

In classical optimization theory, the purpose of induction variable analysis is to strength reduce the calculation of induction variables, or to eliminate the calculation entirely. More recently, with the increasing popularity of vector and parallel processors, induction variable analysis has been used to determine the inherent parallelism in a program. Induction variable analysis, coupled with *forward substitution* [KKLW80,AK87], is used to transform array subscripts into standardized linear functions of loop index variables to facilitate the dependence analysis required for parallelization. These techniques are applicable to the problem of removing constraint checks because, in order to collect accurate data dependence information, it is necessary to determine the subarrays referenced and modified within a loop.

It is natural then to extend these techniques to the elimination of index and range constraint checks within the body of a loop by determining the minimum and maximum values attained by

an induction variable within the loop and using that information to create equivalent constraint checks within the loop preheader. Of course, once the mappings of auxiliary induction variables onto basic induction variables have been determined, induction variable substitution may be applied to transform constraint checks referencing various auxiliary induction variables into checks referencing a single basic induction variable. The structure of the Ada **for** loop makes this transformation particularly inviting. The loop parameter is incremented by one on each iteration of the loop, and may not be modified by the programmer. Also, the direction of the loop is clearly specified in the **for** statement. It is, therefore, particularly easy to convert formulas for induction variables into linear functions of the `'Length` of the loop parameter, as we shall show.

There are three basic components to the algorithm, all operating concurrently. They are:

1. Recognition of basic induction variables and their families of auxiliary induction variables
2. Recognition of constraint checks which are candidates for code motion
3. Creation of equivalent tests in the loop preheader

2.3.1 Recognition of Induction Variables

In this discussion, a *basic induction variable* is any variable defined in a statement of the form $i := i + c;$ for some constant c. An *auxiliary induction variable* is any variable defined in a statement of the form $j := a * i + b;$ where a and b are constants, and i is a basic induction variable. As we will show later, there are no inherent theoretical difficulties in extending the theory to loop invariants. But there are technical problems of symbolic manipulation which need to be addressed. Since most subscript expressions tend to be simple, restricting our attention to constants should not seriously affect performance.

Recognition of induction variables occurs in two disjoint areas. The loop parameter of the **for** loop is identified immediately upon processing the **for** statement. Any other induction variables are recognized when each individual statement in the loop is examined for side effects. A statement of the form $i := i + c;$ where i is of some scalar integer subtype and c is constant causes i to be recognized as a *candidate* basic induction variable. It can only be considered a candidate at this point because i may be redefined at some later point in the loop in a way which violates the strict arithmetic progression of values that the previous statement suggests. However, when i is first recognized as an induction variable, a control block is pushed on to a stack anchored off the symbol table entry for i. This control block indicates that i is a *basic* induction variable, and contains the nesting level of the loop, and the increment c. The control block must be stacked because it is possible that i was already identified as an induction variable in some containing loop.

It is possible that i is redefined later in the same loop. If the later redefinition is of the form $i := i + d_i;$, then the increment in the control block for i is adjusted to $c + d_i$. As an example of this, consider the loop in Section 2.3.3.3 on page 21. i is defined twice in that loop—on lines 2 and 6. The assignment on line 2, $i := i - 3;$, causes the algorithm to mark i as a candidate basic induction variable with an increment of -3. Then, after the second assignment (on line 6): $i := i + 2;$, i's increment is changed to -1 $(= -3 + 2)$.

If, instead, the definition is of any other form, a non-linear function or an **out** variable of a procedure call, for example, then i is invalidated as an induction variable for the remainder of the loop, and all containing loops. All auxiliary induction variables defined based on i are invalidated as well.

When a statement of the form $j := a_j * i + b_j$; is processed, j is identified as an *auxiliary* induction variable if j is a scalar variable of an integer subtype, i is a basic induction variable, a_j and b_j are constants, and j has not already been invalidated as an induction variable. When an auxiliary induction variable is first recognized in a loop, a control block is pushed onto a stack anchored off the variable's symbol table entry. In addition to indicating that the variable is an auxiliary induction variable, the control block contains a pointer to the symbol table entry for i, the multiplier a_j, the increment b_j, and c_i, the current increment of the basic induction variable i (the reason for this is discussed below). Finally, j is added to $family(i)$, the set of auxiliary induction variables based on i.

If j is later redefined in the loop in a statement of the form $j := j + c_j$;, for some constant c_j, then j may still be considered an auxiliary induction variable for the remainder of the loop, but with increment $b_j + c_j$ instead of b_j as originally defined. The only difficulty in doing so is the additional bookkeeping required to make sure that constraint checks generated between any two definitions of j are handled properly. This is further discussed in the next two sections. If j is redefined in any other manner, then we invalidate j as an auxiliary induction variable. This may be viewed as overly conservative. It is possible that j is redefined as an auxiliary induction variable on a different basic induction variable. But such cases should be rare, and are not worth the additional overhead.

We indicated earlier that the current increment of i was saved in the control block for j. This is done to handle auxiliary induction variables based on other auxiliary induction variables rather than basic induction variables. Consider the statement $k := a_k * j + b_k$; where j is the auxiliary induction variable described above. Substituting the value of j into the right hand side, we see that k is in $family(i)$ with multiplier $a_k * a_j$ and increment $a_k * b_j + b_k$, and on each iteration of the loop the value of k is incremented by $a_k * a_j * c_i$, i.e., by a multiple of the increment of i when j is defined. It is of course possible that i is redefined between the definition of j and the definition of k, and we must maintain the correct increment for k. In any event, k is identified as an auxiliary variable in $family(i)$, with appropriate multiplier, increment, and increment of i, just as j was above.

2.3.2 Recognition of Reducible Constraint Checks

This phase occurs after test analysis decides that a particular test is not redundant and may not be eliminated. If the expression being tested is of a form which allows us to determine its lower or upper bound within the loop, based on the type of test, then the test may be discarded from the body of the loop and replaced with an equivalent test on the bound in the preheader of the loop (as demonstrated in the next section). At present, the algorithm only handles expressions of the form $a * i + b$, where a and b are loop constants with known signs, and i is a scalar integer object which has not been invalidated as an induction variable.

We could have restricted our attention to tests which reference variables already marked as possible induction variables, but in so doing we would not be able to handle examples such as

the following:

```
1.  for i in some'Range loop
2.      a(j) := ··· ;
3.      j := j + 2;
4.  end loop;
```

Since this type of construction occurs frequently[2], it seems worth some additional effort.

We handle this case by assuming that any variable used as a subscript, and not previously seen in the loop, is an induction variable and will be incremented later in the loop. We, therefore, mark the variable as a candidate basic induction variable with increment 0, with the expectation that some later assignment statement in the loop will provide the real increment for the variable. For example, in the above loop, we mark j as a candidate basic induction variable, with increment of 0, strictly on the basis of it being used as a subscript in line 2. Then, after line 3 is inspected, j's increment is changed to 2. If the candidate basic induction variable is later defined as an *auxiliary* induction variable, then it is invalidated entirely. The main problem with prematurely marking variables such as j as induction variables in this way is that we may be generating a large number of candidate tests which will be invalidated later. If this proves to be a problem in practice, we can minimize the overhead by extending the prepass, described earlier, to determine which variables defined in the loop cannot be induction variables.

When a candidate test is found, we simply record its existence for later processing. Until the end of the loop is reached, we cannot tell if the candidate basic induction variable, upon which the tested expression is ultimately based, is indeed an induction variable. The information retained for the candidate test includes

1. a pointer to the test just created in the syntax tree
2. the symbol table address of the induction variable in the expression
3. the current increment for the induction variable

The reason for maintaining the current increment for the induction variable is to handle additional definitions of the induction variable which may occur later in the loop. If the induction variable is modified later, and we do not keep track of the initial values, we cannot accurately determine the range of values that it assumes at any point in the loop. This is important because it allows us to handle tests which are created before definitions, such as the range check on j suggested by line 2 of the previous example, or between definitions, such as the tests referencing j in line 4 of the example on page 21. We defer further explanation until we discuss creation of the tests in the preheader. Note, however, that any later change to an induction variable still causes the tests on its uselist to be invalidated by available expression analysis. We do not record the multiplier for an auxiliary induction variable in the record associated with the test because we invalidate the auxiliary induction variable if the multiplier changes. Note also that it is unnecessary to record any information concerning the underlying basic induction variable because this information is recorded in the control block for the auxiliary induction variable.

[2]This construct appears, for example, in the LINPACK package for solving linear equations[Don81].

2.3.3 Creation of Equivalent Tests in the Loop Preheader

By the time the end of the loop is reached, we have collected a list of induction variables defined in the loop and a list of candidate tests which may be replaced with equivalent tests in the prologue of the loop. All that remains then is to process the list of tests and create equivalents for those which reference valid induction variables. We first provide several examples to illustrate the possibilities handled by the algorithm. We must also emphasize that given the current wording of LRM 11.6, the transformations presented in this section are valid *only in the absence of programmer-defined exception handlers.* This restriction will be discussed more fully in Section 2.3.6.

2.3.3.1 Loop Parameter

```
1.  for i in some'Range loop
2.      --ge(4 * i + 3,a'First); le(4 * i + 3,a'Last);
3.      a(4 * i + 3) := ··· ;
4.  end loop;
```

This should be a fairly common case in Ada programs because loop parameters can only change by a unit on each iteration of the loop. The comment indicates the index checks that are required. For example, $ge(4 * i + 3, a\text{'First})$ checks that $4 * i + 3 \geq a\text{'First}$. Since $4 * i + 3$ increases in the loop, the tests within the loop may be eliminated and replaced by

$$ge(4 * (i\text{'First}) + 3, a\text{'First});\ le(4 * (i\text{'Last}) + 3, a\text{'Last});$$

The tedious part of the process is determining whether the expression is increasing or decreasing within the loop, to know whether to replace the loop parameter with its minimum value or its maximum value. Table 1 in Appendix A details the options and correct replacements.

2.3.3.2 Basic Induction Variable

```
1.  for lp in some'Range loop
2.      i := i - 3;
3.      --ge(i,a'First); le(i,a'Last);
4.      a(i) := ··· ;
5.  end loop;
```

This is also a fairly simple case. A test involving a basic induction variable can be replaced with one computing the initial and/or final value of the induction variable. In this case, each iteration of the loop causes i to assume a lower value. The replacement tests therefore are:

$$ge(i - 3 * lp\text{'Length}, a\text{'First});\ le(i - 3, a\text{'Last});$$

In the above equation, as well as in all other examples in this paper, *object*'Length is used to mean the number distinct values in the subtype of *object*. Note that in these replacement tests, i is the original value prior to entry into the loop.

The next example is more difficult.

2.3.3.3 Auxiliary Induction Variable

```
1.  for lp in some'Range in reverse loop
2.      i := i - 3;
3.      j := 4 * i + 2;
4.      --ge(-3 * j,a'First); le(-3 * j,b'Last);
5.      a(-3 * j) := ··· ;
6.      i := i + 2;
7.      j := j + 1;
8.      --ge(j,b'First); le(j,b'Last);
9.      b(j) := ··· ;
10.     k := -3 * i + 4;
11.     --ge(k,c'First); le(k,c'Last);
12.     c(k) := ··· ;
13. end loop;
```

This example illustrates the full range of the algorithm. Admittedly, it is unlikely that such code will be seen often in practice. In any event, there are several areas of difficulty. For one thing, the auxiliary variable j assumes different values for the different constraint checks illustrated. Also, the basic induction variable i is redefined after j is defined, but not after k is defined. By keeping track of these changes, we are able to eliminate all six constraint checks shown from the body of the loop. Let's consider them in reverse order. The tests in line 11 are really no different from those in the previous example. As a result of the two assignments to i in lines 2 and 6, i is decremented by 1 on each iteration of the loop when control passes line 6. Recognizing that k is a decreasing function of i, the constraint checks in line 11 may be replaced by

$$ge(-3 * (i - 1) + 4, c\text{'First}); \; le(-3 * (i + (lp\text{'Length}) * (-1)) + 4, c\text{'Last});$$

Again, i is the value of the i prior to entry into the body of the loop.

Now consider the constraint checks in line 8. The value of j used in line 8 is $j := 4 * i_2 + 3$, where i_2 is the value of i set in line 2. But i has since been redefined and has a different increment at line 8, thus the need to maintain the value of the increment of the basic induction variable at the time the auxiliary induction variable is first defined. Without that, we cannot compute the correct initial and final values of j used at line 9. The tests inserted into the prologue are

$$ge(4 * (i - (lp\text{'Length}) * 3) + 2, b\text{'First}); \; le(4 * (i - 3) + 2, b\text{'Last});$$

The tests in line 4 are different from the other two sets of tests because the value of i is changed later in the loop. The effect of the later adjustment is that the incremental change to i observed at line 3 changes after the first iteration. On the first iteration, i is reduced by 3, but on each succeeding iteration, i is reduced by 1, i.e., the range of values for i at line 3 is

$$i - 3, \; i - 3 - 1 * (1), \; \ldots \; , i - 3 - 1 * (lp\text{'Length} - 1)$$

and the range of values for j is

$$4 * (i - 3) + 2, \ \ldots \ , \ 4 * (i - 3 - 1 * (lp\text{'}\mathtt{Length} - 1)) + 2.$$

It is easy to see now that, because the expression in line 4 is decreasing, the correct replacement tests are

$$ge(-3 * (4 * (i - 3) + 2), b\text{'}\mathtt{First});$$
$$le(-3 * (4 * (i - 3 - 1 * (lp\text{'}\mathtt{Length} - 1)) + 2), b\text{'}\mathtt{Last});$$

The purpose of the above examples is to clarify the type of analysis required at the end of the loop. After the body of the loop is processed, the list of candidate tests is processed. If the referenced induction variable has been invalidated, then the processing immediately proceeds to the next candidate test. So assume now that the referenced induction variable is still valid at the end of the loop.

If the referenced variable is the loop parameter, lp, then the new test for the prologue is created by simply substituting either lp'`First` or lp'`Last` into the test for lp, depending on the type of test, slope of the function being tested, and whether lp is increasing or decreasing in the loop.

If the referenced variable is a basic induction variable, i, we first check whether c_1, the increment of i when the test was created, is the same as c_n, the increment at the end of the loop. If so, then i changes by the same amount on each iteration of the loop. The new test is created by substituting for i either $i + c_1$ or $i + (lp\text{'}\mathtt{Length}) * c_1$, depending on the type of test, the slope of function being tested and $sign(c_1)$. If $c_1 \neq c_n$, then i is incremented by c_1 on the first iteration of the loop, and c_n on each succeeding iteration. The new test is therefore created by replacing i with either $i + c_1$ or $i + c_1 + (lp\text{'}\mathtt{Length} - 1) * (c_n)$, again depending on the items mentioned above.

If the referenced variable is an auxiliary induction variable, j, the processing is similar to that described in the previous paragraph. It is not important whether or not the increment for j is changed later (except for our ability to easily recreate the definition of j) because any later change has no impact on the value of j at the test. But j is a function of i, so it is essential to understand i's behavior in the loop. Assuming then that $j := a_j * i + b_j;$, the new test for the loop prologue is created by substituting j with either $a_j * (i + c_1) + b_j$ or $a_j * (i + (lp\text{'}\mathtt{Length}) * c_1) + b_j$ if $c_1 = c_n$. Otherwise, the choices for j's replacements are $a_j * (i + c_1) + b_j$ or $a_j * (i + c_1 + (lp\text{'}\mathtt{Length} - 1) * c_n) + b_j$.

The exact option to choose is detailed in Table 3 in Appendix A.

As the final step in the algorithm, the control blocks attached to the symbol table are removed and, in the case of 1-trip loops, relevant information obtained during induction variable analysis in the current loop is retained for possible use in the remainder of the enclosing block. Because we cannot tell at compile-time whether any 0-trip loop is executed or not, any definition of a variable identified as a possible induction variable in any outer loop must be treated as an invalidation of the variable. Note that this lack of knowledge does not prevent us from creating the equivalent tests in the preheader of the current loop.

If i is a basic induction variable with increment c_1 in the current loop, the number of iterations is known, and i has not already been invalidated in the containing loop, then the loop can be considered as establishing i as a basic induction variable in the outer loop, or

updating the definition of an induction variable in the outer loop. For example, consider the following loops:

```
1.  for i1 in some'Range loop
2.      i := i + 1;
3.      for i2 in 1..10 loop
4.          i := i + 2;
5.          a(i) := ··· ;
6.      end loop;
7.      b(i) := ··· ;
8.  end loop;
```

The effect of the $i2$ loop is equivalent to the assignment statement $i := i + 20$; and the increment of i at the end of the outer loop is 21.

After the entry for a basic induction variable is adjusted, each of the auxiliary induction variables is processed. It is interesting to note that since an auxiliary induction variable is a function of some other variable, unlike a basic induction variable that may be viewed as an "adjustment," an auxiliary induction variable in an inner loop can only be an auxiliary induction variable in the family of the same basic induction variable in the outer loop as in the inner loop. It is unlikely that this will prove beneficial very often, so we choose to simply invalidate the variable as an induction variable in any outer loop.

Finally, any variable invalidated in the loop is invalidated in all outer loops.

2.3.4 Restrictions

We have already mentioned (and later discuss further) that the induction variable algorithm is valid only in the absence of exception handlers. The algorithm also assumes that there are no **exit** statements, subroutine calls, or conditional statements within the loop. With the presence of an **exit** in the loop, it is impossible for us to accurately compute the number of iterations for the loop. As a result, it is impossible for us to create tests that accurately compute the extrema of any induction variable within the loop. By incorrectly basing a test on a value that cannot actually be obtained, we might raise an exception in the loop preheader that does not occur in the original program.

The restriction against subroutine calls is a little subtler, but is again concerned with illegally raising an exception in the loop prologue. In this case, the concern is that the subroutine call results in an unhandled exception that is propagated back to the loop. Raising some other exception would be a breach of section 11.6 of the LRM which states that "...an implementation may only use an alternative order if it can guarantee that the effect of the program is not changed by the ordering."

The restriction against conditional statements in the loop is primarily technical rather that theoretical. Given the recursive nature of the algorithm, it seems like an unnecessary complication at this point to keep track of candidate induction variables in the presence of conditional statements, where the original state of the loop must be restored after every conditional block. A less drastic approach that might be considered in the future is to simply invalidate any variable that is assigned in a conditional statement. This will require additional flags to indicate whether or not a given linear block is contained within a loop and/or a conditional statement.

2.3.5 Extensions

The present algorithm finds induction variables only when the multiplier and increment are constant. An obvious extension is to include those cases where the multiplier and increment are loop constants whose *signs* are known. After all, the critical information is knowing whether the variable is increasing or decreasing in the loop, not the actual value that it assumes. Furthermore, the information is already available from the loop prepass and constant propagation phase. The primary problem with relaxing the requirements in this way is knowing whether the induction variable is increasing or decreasing at any point in the loop. The obvious approach to solving this problem is to allow changes only when the sign of a new increment matches the sign of the current increment. Another difficulty in extending the algorithm in this way is the overhead of maintaining the information. Instead of computing the current multiplier and increment as we progress through the loop, and saving each in a single word in the appropriate control blocks, we would have to build tree segments for the actual computation. It is unlikely that the time and space required to maintain the information will yield sufficient return.

A second extension is removal of the restriction against conditional statements in **for** loops. Conditional statement processing could be modified to invalidate any scalar integer variable that is defined in any conditional block.

2.3.6 Legality of the Transformation

Presently, there is considerable disagreement in the Ada community over the validity of code motion in Ada programs. Brender [Bre87], for example, states

> The many comments collected in AI-00315[3] make it clear that there is considerable confusion in the Ada community regarding the intent and implications of section 11.6 of the Ada manual.

Dewar and Hilfinger [DH87] have proposed that LRM 11.6 be eliminated entirely, for the reasons that "...the set of allowed transformations is ill-defined," and that is has proven "less than satisfactory" in informing implementers of what actions they may take to enhance object code performance.

A major source of the concern raised by LRM 11.6 is the following statement:

> Additional freedom is left to an implementation for reordering actions involving predefined operations that are either predefined operators or basic operations *other than assignments* (italics mine).

It is clear that the implications for vector and parallel architectures were not considered when this statement was written. For example, loops such as the following,

```
1.  for i in 1..n loop
2.      a(i) := b(i) + c(i);
3.  end loop;
```

cannot be vectorized for fear that an exception might occur during execution. For if the exception occurs during iteration i', the language requires that $a(1), \ldots, a(i'-1)$ *must* be

[3] Ada Issue Reports related to LRM 11.6.

changed while $a(i'), \ldots, a(n)$ must remain unchanged—something very difficult to accomplish on a vector machine. Similarly, the above restriction hinders exploitation of parallel processors in the following way. Ordinarily, we want two independent assignment statements such as

```
a := b + c;
d := e + f;
```

to execute concurrently on a parallel processor. But if an implementation does this, it may be violating the standard because it cannot guarantee that the assignment to a will complete before the assignment to d.

The transformation described in this section raises similar issues. This is not surprising when we consider that it is based on optimizations used by vectorizing compilers. First note that no test is replaced by an equivalent test in the prologue unless we can prove that the loop executes the number of iterations specified in the **for** statement. Thus, in a correct program, there will be no discernible difference in the output of any program to which this transformation is applied. That may not be the case for an incorrect program. One problem is that since checks are moved to the loop prologue, exceptions will be raised before any assignments in the loop are performed. Another problem is that the transformation has the effect of reordering the constraint checks in the loop. Once we move any test out of a loop which contains multiple run-time exceptions, there is no way to guarantee that the exception raised in the prologue will be the same one that would be raised in a *canonical* execution of the program (LRM 11.6). On the contrary, it is extremely unlikely that the 2 programs will terminate with the same exception.

In response to the confusion raised by LRM 11.6, [Bre87] proposes that in the absence of an exception handler, the implementer be allowed to perform all possible optimizations under the assumption that no exceptions will occur, and leave undefined the results of the program should an exception occur. This proposal appears to have gained acceptance by the Ada community, and allows us to claim that our transformation is valid in the absence of exception handlers.

3 Effects of Tasking and Shared Variables

So far, we have discussed this algorithm only in the context of a sequential program. But Ada is designed to support real-time systems, and one of its advances over other high-level languages is its repertoire of high-level multiprocessing constructs. *Tasks* are Ada processes which execute in parallel. Tasks contain two control structures not previously addressed in this paper: the **entry call** and the **accept** statement. Two tasks *synchronize* when one calls the other with an **entry call**, and the other accepts the call at an **accept** statement. Tasks communicate in much the same way as subroutines do in sequential programs — via parameters specified in the **entry call** and/or variables which are visible to both tasks. The standard scope rules that apply to sequential programs apply to tasks as well. Tasks executing concurrently are allowed to reference and change any scalar or access variable declared in some enclosing frame. Any variable that can be accessed by multiple tasks is called a *shared variable*. This section discusses how tasking and the presence of shared variables affect our algorithm.

The Ada rules concerning shared variables are intended to support concurrent reads and exclusive writes (CREW). If a task reads or writes a variable between two synchronization

points[4], the compiler may assume that the variable is not changed by any other task during that time period. Any program which fails to obey those two assumptions is considered *erroneous*. The language also provides the **pragma** SHARED to allow the programmer to explicitly specify which scalar and access variables are referenced by multiple tasks. The programmer may use shared variables so specified with greater freedom than allowed for other shared variables because *any* reference is defined to be a synchronization point, and automatically satisfies the above assumptions. The effect of this freedom is that the compiler may make no assumptions at all about the value of a variable specified as SHARED. The effect on the algorithm in this paper is disastrous; no test which references such a variable is valid at any point other than the point at which it is created. There is therefore no reason even to include tests which reference SHARED variables in any of the analysis. All that can be done is to simply insert them into the syntax tree and go on to the next statement. Fortunately, matters are not so hopeless for implicitly shared variables (i.e., those not explicitly specified in a **pragma** SHARED statement). While the assumptions in LRM 9.11(4-5) may be incomplete and pose subtle problems for storage management and code generation [Shu87], they are sufficient to allow the algorithm to work properly, if somewhat more conservatively.

First, let us consider how these rules affect the available expression analysis portion of the algorithm. In this phase, a test is eliminated at a point P when it can be proven that the test was performed at a dominator of P, and has not since been invalidated by an assignment to any of the operands it references. Ignoring the **accept** statement for a moment, tasking does not change the control flow *within* a given procedure, so the data flow equations presented earlier are valid in both sequential and parallel executions. But tasking certainly does affect what we may safely assume about the value of any object accessible to another program. In a sequential procedure, we are guaranteed to have total control of every variable visible to the procedure; no variable may be modified until the procedure gives up control to some other procedure. In tasks, we lose unilateral control of any shared variable. But because of the CREW philosophy, we have not lost all.

LRM 9.11 allows a local copy of any shared variable not explicitly named in a SHARED **pragma** to be maintained and used until a synchronization point is reached. For a shared variable not changed by a task, using the local copy is equivalent to referencing the actual variable. For a shared variable changed by a task, the local copy *must* be used until either it is stored in the shared variable, or a synchronization point is reached. If use of a local copy in this manner causes the program to yield different results, the program is erroneous. Since a routine certainly has unilateral control over any local variable, these assumptions indicate that it is safe for the algorithm to act, *between any two synchronization points*, as if it has unilateral control of any shared variable not referenced in a SHARED **pragma**. In particular, the algorithm is safe for tasks as well as for sequential programs.

Let's outline the control flow of a task to show why this is so. At its start, a task T is synchronized with the task that caused its activation. From the time it begins execution to the first synchronization point, it may legally assume that no shared variable either changed or just referenced by T has been changed by any other task, and that all valid constraint checks

[4] According to the LRM, "Two tasks are synchronized at the start and at the end of their rendezvous. At the start and at the end of its activation, a task is synchronized with the task that causes this activation. A task that has completed its execution is synchronized with any other task."

remain valid. Suppose that the first synchronization point reached is an **entry call**. Other than causing T to be suspended, the **entry call** is similar to a **procedure call**, and is handled in an identical fashion. All **out** and **in out** parameters, as well as all non-local variables receive new store numbers and new lattice values of *bottom*. All tests based on those variables are invalidated. Nor does it matter if the statement is either a timed **entry call** or a conditional **entry call**. Both cases are handled in the same manner as a **procedure call** in a conditional statement. Constraint checks valid prior to the **entry** statement are still considered valid in the conditional block associated with the **delay** statement in the timed **entry call**, and in the conditional block associated with the **else** statement in the conditional **entry call**. But no constraint check killed by any conditional block is ever valid after any form of conditional statement.

Now suppose that the first synchronization point is an **accept** statement. There is no exactly comparable Ada construct in sequential programs. Though similar to a called procedure, the **accept** statement most resembles the **entry** statement in FORTRAN and PL/I. We know that the caller is now suspended, but that is of no importance to us; T still maintains control. As in the case of any called procedure, the lattice values of any **in** and **in out** parameters are set to *bottom*. The parameters are local to the **accept** statement, so there are no existing constraint checks that are affected. The lattice values of any *unreferenced* non-local variables are reset as well. Since they have not been referenced in T, it is perfectly legal for them to have been changed in some other task. But, existing constraint checks are still valid, and may be safely used within the **accept** statement. What is more interesting is what happens *after* the **accept** statement.

As we've stressed several times above, the LRM allows the algorithm to process any shared variables referenced between two consecutive synchronization points without change. But what about *un*referenced variables between two synchronization points? We may make no assumption about them, and in fact we must assume that they have been changed. If the last synchronization point was an **entry** statement, then all tests were killed at the **entry** statement and no additional processing is required. But if the last synchronization point is the end of an **accept** statement, then additional processing is required; all variables unreferenced up to the next synchronization point must be killed, and all constraint checks on their uselists invalidated.

To do this, an additional piece of information is necessary for every object — a *referenced flag* to be used in the following way. Starting from the beginning of the compilation unit, the first synchronization point of any task, all variables referenced have their referenced flags set. This process continues until the **end** of an **accept** statement is reached. By the assumptions, we are assured that our information about referenced variables is correct, and that any actions taken by the algorithm on the basis of that information are safe. But we know nothing about the unreferenced variables. Therefore, all tests which use any previously unreferenced variable must be killed at this point. Of course, when we start from the beginning of the compilation unit, there will be no tests associated with unreferenced variables. But some of those referenced up to the first synchronization point encountered may not be referenced in the code up to the next one and would have to be killed. Finally, the reference flags of all variables are reset to prepare for any **accept** statements which follow.

There is no additional difficulty when considering sequences of statements or conditional statements, including selective wait statements. The **end** of the **accept** statement simply

represents one more point where tests are killed, and is easily handled by the algorithm. Loops, however, complicate matters once again. In the sequential case, a test created outside the loop is valid at the top of the loop if none of the variables it references are changed within the body of the loop. In the parallel case, where a loop contains an **accept** statement, such a test is valid at the top of the loop only if *all* variables referenced in the test are later referenced on *every* path through the loop. For, as mentioned above, if any variable is not mentioned on a path between two synchronization points, say the **end** of the **accept** on iteration i and the **accept** on iteration $i+1$, we must assume that it has been changed by some other task between these two points, and cancel any test on the variable's uselist created outside the loop. Because a variable must be referenced on every path to be safe for our purposes, we are forced to perform data flow analysis on the loop to accurately determine which variables are safe. Given the philosophy of the algorithm presented here, that approach is unacceptable.

A cheaper and more conservative approach is to simply kill all tests associated with non-local variables upon entry into any loop which contains an **accept** statement. Ultimately, this might not yield poorer results than that obtained by the full data flow analysis. It is unlikely that many variables are used on every path. Another approach is to scan the loop to determine its control structure. If the loop does not contain any conditional statements to complicate the control flow, then it is simpler to determine the set of unreferenced variables.

4 Exception Handling

We have already discussed in Section 2.3.4 how code motion by induction variable analysis is curtailed by the possibility of exceptions propagated by called subroutines. We now turn our attention to how available expression analysis and constant propagation are affected.

4.1 Exception Handlers

First let us consider the exception handler itself. Theoretically, the handler can be invoked from any point in the containing frame. Thus, it is impossible to determine what constraint checks are valid, or the value of any object upon entry into the constraint handler. All tests must be invalidated, and all objects must have their lattice values set to *bottom*. However, there is no reason why processing cannot begin anew within the handler. The exception handler may be viewed as a new procedure invoked without any parameters. It is questionable whether anyone would want to do so. It is assumed that the exception handler is used only for exceptional cases and so any optimization would go unnoticed.

4.2 Frames Containing Exception Handlers

Now let us consider the impact of programmer-defined exception handlers on a frame processed by the algorithm.

Ignoring these handlers has allowed us to completely ignore the transfer of control that results from a failed constraint check. In the absence of user-defined handlers, the net effect of an exception is that execution of the program is abandoned. Thus, a given statement in the program can be executed only if no exception has been raised at some earlier point. If the user provides exception handlers, we can no longer make that assumption.

We see, then, that the real problem "raised" by user-defined exception handlers, as they relate to this algorithm, is the transfer of control that they represent. This transfer of control can be initiated in basically two ways. We have already seen that the compiler will raise an exception when a constraint check fails. The language also contains the **raise** statement to allow the programmer to explicitly signal an exception and transfer control to the corresponding handler. The **raise** statement may raise both predefined and declared exceptions; the programmer is not required to provide a handler in either case. But an explicit **raise** is no more difficult to handle than an implicit **raise** contained within a constraint check. If a **raise** statement raises an exception which has no corresponding handler anywhere in the program, the situation is no different from what we already considered. Conversely, if the user provides an exception handler for a predefined exception such as *constraint_error*, our basic assumption is no longer valid.

This further illustrates that it is the transfer of control that is important to us and not the precise source of the transfer. In the remainder of this section then, we focus on the user-defined exception handler and the transfer of control that it represents, rather than the **raise** statement which simply initiates the transfer.

4.2.1 Available Expression Analysis

If the program is composed of a single frame, then the existence of a exception handler has absolutely no effect on either algorithm. Because a test is only eliminated when it is dominated by an identical test, we are assured that the exception is processed by the appropriate handler.

We cannot be so complacent about exception handlers in frames embedded in other frames. Consider the following example.

```
1.  begin
2.      ...
3.      begin
4.          a(i) := ...;
5.          ...
6.      exception
7.          when INDEX_CHECK ⇒
8.      end;
9.      ...
10.     a(i) := ...;
11. end;
```

The present algorithm makes the implicit assumption that it is safe to eliminate a redundant check because control could not reach this point if the previous check had raised an exception. This assumption is violated in the above example. Ada views exceptions as terminating events and provides the exception handler as a facility for local termination of frames upon detection of errors [IBFW86, pp. 311-312].

Suppose that the references to $a(i)$ in lines 4 and 10 each raise index_check. The first index_check, raised at line 4 within the inner frame, will be handled by the programmer's exception handler. If the constraint check for $a(i)$ at line 10 is removed because the earlier one dominates it, then the generated code will be in error.

But there is a second, more serious, problem which forces us to be even more conservative. This second problem is similar to the one presented by **exit** statements. We have no way of knowing at compile time when the exception handler will be invoked. Even if the programmer invokes the handler explicitly with a **raise** statement, the **raise** statement will almost always be contained in a conditional clause. We can never, therefore, be certain of how much of the frame executes before control is returned to the enclosing frame, and so we can never prove that *any* of the constraint checks (except for the first) in the inner frame are executed. Thus, we can never prove that any constraint check in an *enclosing* frame is redundant.

The simplest, and probably the best, solution to the problem is to kill all tests generated within a frame containing an exception handler upon exit from that frame. Another possibility is to kill only those tests which follow the first point in the frame where a handled exception may be raised. This first point may be an explicit **raise**, a test generated by the compiler, a subroutine call, or any of them in a nested frame which does not contain its own handler for the same exception. But it is questionable whether this approach will, in practice, provide better results than the first. It can do so only when the frame containing the exception handler *and* its enclosing frame are sufficiently large that a significant number of checks in the portion of the outer frame *following the inner frame* may be eliminated.

It is unlikely that exception handlers will be used in such a way. It is more likely that they will either be used in (1) very small frames created precisely to monitor a particular operation, such as a **read**, or (2) subprograms to clean up after some catastrophic error occurs and further propagate the exception to the caller. In the first case, the frame will likely be too small to cause many tests to be generated. It is, therefore, unlikely that many tests in the remaining portion of the enclosing frame will be found to be redundant. In fact, it is more probable that tests in the inner frame will be eliminated because of identical tests in the preceding portion of the enclosing frame. In the second case, the presence of the handlers really has no effect because all tests are always killed upon reaching the end of a procedure.

4.2.2 Constant Propagation

Not surprisingly, exception handlers increase the difficulty of constant propagation in the same manner as the **exit** statement. The value of a variable upon exit from a frame is the meet of its value at each point in the frame where a handled exception can be raised, and its final value upon exit from the frame.

This can be handled in the same manner as for **exit** statements, but given our comments on the likely use of exception handlers, it is questionable whether such efforts would be beneficial. It is probably most efficient to simply set the lattice values of all variables defined in a frame with an exception handler to *bottom* upon exit from the frame.

5 Examples

5.1 Example 1 - Matrix Multiplication

An example commonly used in the literature ([MCM82,Wel78] for example) to illustrate the effectiveness of range checking algorithms is matrix multiplication. Below is an Ada version of the algorithm, with implicit constraint checks encoded as comments. The checks are of

the form *check_type*(*expression*, *constraint*), where the check types are *ge* and *le* for *greater than or equal to*, and *less than or equal to* respectively. Constraint checks which cannot be eliminated are capitalized.

```
procedure matrix is
    n: constant Integer := 50;
    s: Float;
    type square is array(1..n,1..n) of Float;
    a,b,c: square;
begin
    for i in 1..n loop
        --ge(i,1),pseudo; le(i,50),pseudo;
        for j in 1..n loop
            --ge(j,1),pseudo; le(j,50),pseudo;
            s := 0.0;
            for k in 1..n loop
                --ge(k,1),pseudo; le(k,50),pseudo;
                --ge(i,square'First(1)); le(i,square'Last(1));                (1)
                --ge(k,square'First(2)); le(k,square'Last(2));                (1)
                --ge(k,square'First(1)); le(k,square'Last(1));                (1)
                --ge(j,square'First(1)); le(j,square'Last(1));                (1)
                s := a(i,k)*b(k,j) + s;
            end loop ;
            --ge(i,square'First(1)); le(i,square'Last(1));                    (1)
            --ge(j,square'First(1)); le(j,square'Last(1));                    (1)
            c(i,j) := s;
        end loop ;
    end loop ;
end matrix;
```

(1) Simple range analysis

In this case, all tests are easily eliminated by a simple comparison to the constant array bounds. Now consider the same algorithm coded in a package designed for arrays of arbitrary size:

```
package pak1 is
    type square is array(Integer range <>, Integer range <>) of Float;
    procedure matrix_multiply(a,b: square; c: out square);
end pak1;

package body pak1 is
    procedure matrix_multiply(a,b: square; c: out square) is
        n: Integer;
        s: Float;
    begin
```

```
        for i in a'Range(1) loop
            -- ge(i,a'First(1)),pseudo; le(i,a'Last(1)),pseudo;
            for j in b'Range(2) loop
                -- ge(j,b'First(2)),pseudo; le(j,b'Last(2)),pseudo;
                s := 0.0;
                for k in a'Range(2) loop
                    -- ge(k,a'First(2)),pseudo; le(k,a'Last(2)),pseudo;
                    -- ge(i,a'First(1)); le(i,a'Last(1));                    (2)
                    -- ge(k,a'First(2)); le(k,a'Last(2));                    (2)
                    -- GE(K,B'FIRST(1)); LE(K,B'LAST(1));                   (3)
                    -- ge(j,b'First(2)); le(j,b'Last(2));                    (2)
                    s := a(i,k)*b(k,j) + s;
                end loop;
                -- GE(I,C'FIRST(1)); LE(I,C'LAST(1));                       (4)
                -- GE(J,C'FIRST(2)); LE(J,C'LAST(2));                       (5)
                c(i,j) := s;
            end loop;
        end loop;
    end matrix_multiply;
end pak1;
```

(1) Simple range analysis

(2) Redundant

(3) Replaced by ge(k'First,b'First(1)); le(k'Last,b'Last(1)); inserted into prologue of k-loop by induction variable analysis

(4) Loop invariant; moved to prologue of j-loop

(5) Replaced by ge(j'First,c'First(1)); le(j'Last,c'Last(1)); inserted into prologue of j-loop by induction variable analysis

In this case, we are still able to remove all checks from the innermost loop, even though the bounds of the array are unknown. We are unable to totally eliminate the tests against *b*'First and *b*'Last because there is no way to prove that the matrices are square as the programmer expects. Because the arrays are parameters, we could not assume that they shared a common template and code all tests in terms of a single template. Similarly, the range checks on the subscripts of *c* could not be eliminated completely. Of course, the results are this good only because the **for** statements were coded in a somewhat intelligent fashion. If the programmer coded them all with the same range, a'Range(1) for example, most of the tests would have remained. But we would have still been able to move the tests out of the loop because they referenced only induction variables.

Finally, consider the following revision of the example which takes a more active role in safeguarding itself from invalid input:

```
package pak1 is
    subtype smallint is Integer range 1..100;
    type square_array is
        array(Integer range<>, Integer range<>) of Integer;
    type square(lower, upper : smallint) is record
        data:square_array(lower..upper, lower..upper);
    end record;
    function "*"(a,b : square) return square;
end pak1;

package body pak1 is
    function"*"(a,b : square) return square is
        n: Integer;
        s: Integer;
        c: square(a.lower,a.upper);
    begin
        if a.lower = b.lower and a.upper = b.upper then
            for i in a.data'Range(1) loop
                --ge(i,a.lower),pseudo; le(i,a.upper),pseudo;
                for j in b.data'Range(2) loop
                    --ge(j,b.lower).pseudo; le(j,b.upper),pseudo;
                    s := 0;
                    for k in a.data'Range(2) loop
                        --ge(k,a.data'First(2) = a.lower),pseudo;
                        --le(k,a.data'Last(2) = a.upper),pseudo;
                        --ge(i,a.data'First(1) = a.lower);                (2)
                        --le(i,a.data'Last(1) = a.upper);                 (2)
                        --ge(k,a.data'First(2) = a.lower);                (2)
                        --le(k,a.data'Last(2) = a.upper);                 (2)
                        --ge(k,b.data'First(1) = b.lower);                (3)
                        --le(k,b.data'Last(1) = b.upper);                 (4)
                        --ge(j,b.data'First(2) = b.lower);                (2)
                        --ge(j,b.data'Last(2) = b.upper);                 (2)
                        s := a.data(i,k) * b.data(k,j) + s;
                    end loop ;
                    --ge(i,c.data'First(1) = c.lower);                    (5)
                    --le(i,c.data'Last(1) = c.upper);                     (6)
                    --ge(j,c.data'First(2) = c.lower);                    (7)
                    --ge(j,c.data'Last(2) = c.upper);                     (8)
                    c.data(i,j) := s;
                end loop ;
            end loop ;
            return c;
        end if;
```

```
    end "*";
end pak1;
```

(1) Simple range analysis

(2) Redundant

(3) Replaced by ge(a.data'First(2),b.data'First(1)) by induction variable analysis.

(4) Replaced by le(a.data'Last(2),b.data'Last(1)) by induction variable analysis.

(5) Replaced by ge(a.data'First(1),c.data'First(1)) by induction variable analysis.

(6) Replaced by le(a.data'Last(1),c.data'Last(1)) by induction variable analysis.

(7) Replaced by ge(b.data'First(2),c.data'First(2)) by induction variable analysis.

(8) Replaced by le(b.data'Last(2),c.data'Last(2)) by induction variable analysis.

In this revision, the algorithm is coded as a function to control the shape of the output. Also, *square* has been specified as a discriminated record to guarantee that the input to "*" are indeed square arrays. Finally, the ranges of the two input arrays are checked for equality before any multiplication occurs. Unfortunately, the results are no better in this case than those of the previous example, despite the additional information. In fact, the additional information is not used at all. What is needed is a means of associating constraint checks based on different attributes, of the same or different variables, when those attributes are known to be equal.

It might first appear that this could easily be accomplished by simply using value numbers, rather than the weaker store numbers. In fact, value numbers would be sufficient to eliminate the remaining constraint checks in the above example. Value numbers' deficiency lie in that they can only associate tests created *after* the point where it is learned that the referenced attributes are equal. In the above example, all processing takes place within the sequence of statements guarded by the **if** statement, where it is known that a and b have the same shape, and after c is declared to have the same shape as a. A preferable solution would be to group valid constraint checks into equivalences classes, or *pools*, such as those suggested by [Kil73] for handling available expressions, once the referenced attributes are known to be equal. Because it works on tests already created, it must also have the capability of separating equivalence classes once the scope of a guard is exited. For example, in "*" above, the constraint checks for *a.data*'First(1) and *b.data*'First(1) can be associated within the **then** part of the **if** statement, but must remain separate elsewhere.

5.2 Example 2 - Binary Search

Another example from the literature is binary search. Below is an Ada version of the algorithm appearing in both [Wel78] and [SI77], annotated as above.

```
package pak1 is
    type table is array(1..100) of Integer;
    subtype mid is Integer range 0..100;
```

```
end pak1;

with pak1; use pak1;
procedure binarysearch1(A: in out table; key: Integer;
                        middle: in out mid) is
    subtype low_type is Integer range 1..101;
    low: low_type;
    high: mid;
begin
    --ge(100,mid'First); le(100,mid'Last);                                  (1)
    high := 100;
    --ge(1,low_type'First); le(1,low_type'Last);                            (1)
    low := 1;
    while low <= high loop
        --GE( (LOW+HIGH)/2, MID'FIRST);
        --LE( (LOW+HIGH)/2, MID'LAST);
        middle := (low+high) / 2;
        --GE(MIDDLE, TABLE'FIRST);
        --le(middle, table'Last);                                           (1)
        if a(middle) = key then
            --ge(high+1, low_type'First); le(high+1, low_type'Last);        (1)
            low := high + 1;
        elsif
        --ge(middle, table'First);                                          (2)
        --le(middle, table'Last);                                           (1)
        a(middle) > key then
            --GE(MIDDLE-1,MID'FIRST);
            --le(middle-1,mid'Last);                                        (1)
            high := middle -1;
        else
            --ge(middle+1,low_type'First); le(middle+1,low_type'Last);     (1)
            low := middle + 1;
        end if;
    end loop;
    --ge(middle, table'First);                                              (2)
    --le(middle, table'Last);                                               (1)
    if a(middle) /= key then
        --ge(0, mid_type'First); le(0, mid_type'Last);                      (1)
        middle := 0;
    end if;
end binarysearch1;
```

(1) Simple range analysis

(2) Redundant

In this case, Welsh reports retaining three partial subscript checks as well as the lower bound check for *high* := *middle* − 1;. The problem is that *middle* is used to return a failure return code if the key is not found and so has a range which is a superset of the range of subscripts for type *table*. Although we cannot eliminate the test for the assignment, we are able to eliminate two of the three subscript checks. The test in the **if** statement is considered part of the containing linear block and so is available when *middle* is again tested in the **elsif** condition. Furthermore, since the **while** loop is a 1-trip loop, as determined by constant propagation, the subscript check is allowed to live after the loop is exited.

As above, let us consider the routine as a more general subroutine capable of handling arrays with arbitrary subscripts.

```
1.  package pak1 is
2.      type table is array(Integer range<>) of Integer;
3.      procedure binarysearch2(A: in out table; key: Integer;
                    middle: in out Integer);
4.  end pak1;

5.  package body pak1 is
6.      procedure binarysearch2(A: in out table; key: Integer;
                    middle: in out Integer) is
7.          subtype lo_range is Integer range a'First..(a'Last + 1);
8.          subtype hi_range is Integer range (a'First- 1)..a'Last;
9.          low: lo_range;
10.         high: hi_range;
11.     begin
            -- GE(A'LAST,HI_RANGE'FIRST);
            -- le(a'Last,hi_range'Last);                              (1)
12.         high := a'Last;
            -- ge(a'First,lo_range'First);                            (1)
            -- LE(A'FIRST,LO_RANGE'LAST);
13.         low := a'First;
14.         while low <= high loop
15.             middle := (low + high) / 2;
                -- GE(MIDDLE,A'FIRST); LE(MIDDLE, A'LAST);
16.             if a(middle) = key then
                    -- GE(HIGH + 1, A'FIRST); LE(HIGH + 1,A'LAST);
17.                 low := high + 1;
18.             elsif
                -- ge(middle,a'First); le(middle, a'Last);            (2)
19.             a(middle) > key then
                    -- GE(MIDDLE − 1, A'FIRST);
                    -- le(middle − 1, a'Last);                        (2)
20.                 high := middle − 1;
21.             else
                    -- ge(middle + 1, a'First);                       (2)
```

```
                    -- LE(MIDDLE + 1, A'LAST);
22.                 low := middle + 1;
23.             end if;
24.         end loop;
            -- GE(MIDDLE, A'FIRST); LE(MIDDLE, A'LAST);
25.         if a(middle) /= key then
26.             middle := 0;
27.         end if;
28.     end binarysearch2;
29. end pak1;
```

(1) Semantic processing

(2) Redundant

Not surprisingly, more tests remain than when all objects had constant ranges. Only six tests were eliminated in the current program whereas 19 were eliminated in the previous example. Most of the additional tests are a result of the possibility that *a* could be a null array. For example, consider the subscript check in line 25. In the last example this check could be eliminated due to the identical test in the **while** loop. But in this case, the algorithm could not guarantee that the loop would be entered and was forced to invalidate the test within the loop when the loop was exited. (In fact, this check will raise a constraint_error in the event that *a* is null.) But the algorithm was able to eliminate some tests nevertheless. The test $le(middle - 1, a\text{'Last})$ in line 20 was eliminated because it was obviated by the test in line 19. The lower bound test in line 21 was eliminated for the same reason. Two tests were generated as a result of the assignment to *low*. This statement causes the loop to be exited when *key* is found in the array. The two tests could have been eliminated if an exit statement had been used instead, without preventing other tests from being eliminated in other parts of the program.

It's also interesting to note that the results would have been better had *low* and *high* been typed as `Integer`, rather than some subtype. In that case, all 10 checks generated due to assignments could have been eliminated (assuming that the hardware would automatically generate interrupts on integer overflow and underflow). This contradicts the intuitive assumption that additional range information will always allow the compiler to eliminate more run-time checking at compile-time.

6 Data Structures

This section describes the data structures used by the test elimination algorithms.

6.1 Constraint Checks

In addition to the text of the constraint check, which is represented as a binary operation in the tree, are a set of status flags to indicate the validity of the check at any point during processing.

type constraint_check **is record**

```
    check_type: (ge, le, equal);  -- All checks are transformed into a
                                  -- check against a particular constraint
    expression: ast_node;         -- pointer to expression within syntax tree
    field: symtab;                -- symbol table entry for template field
    store#: store_num;            -- highest store number of any variable in
                                  -- expression
    level: Positive;              -- creation level of test
    kill: Boolean;                -- true ⇒ check no longer valid, and cannot
                                  -- be used to eliminate newly generated tests
    prolog: Boolean;              -- true ⇒ no object referenced in the check is
                                  -- defined in the current loop
    pseudo: Boolean;              -- true ⇒ ignored by the code generator
    simple: Boolean;              -- true ⇒ expression is simple, look for existing
                                  -- check which covers a superset of range covered
                                  -- by this constraint check
end record;
```

The lists of these checks have the following format. An individual list item is defined as a variant record so that the list returned after processing a linear block can contain both constraint tests and variable definitions. Variable definitions are represented by store numbers.

```
subtype store_num is Integer range −1..max_store_num;

type list_of_items is access list_item;
type item_type is (constraint_check, variable_definition);
type list_item(item: item_type) is record
    next: list_of_items;
    visited: Boolean;  -- used to compute intersections of lists
    case item is
        when constraint_check ⇒
            cc: node;  -- pointer to constraint check in AST
        when variable_definition ⇒
            var_def: store_num;
    end case;
end record;
```

6.2 Store Numbers

Each positive store number maps an object into a tuple consisting of its lattice level, previous store number, and pointer to the object defined. The lattice level of *top* is represented by -1 and is never actually mapped in the array below.

```
type value is record
    level : (top, constant, sign, bottom);
    actual: scalar_value;
    end record;
```

```
type store_num_table_entry is record
    value_data: value;
    variable: symtab;      -- variable defined by store number
    previous: store_num; -- previous value for variable
end record;
store_num_table: array(1..store_num'Last) of store_num_table_entry;
```

6.3 Symbol Table

In most cases, the symbol table entry for a constrained subtype (or equivalently, its template) is the anchor for the list of tests against individual constraints. Because unconstrained objects can assume arbitrary subtypes, it is impossible to associate a single subtype with them. Therefore, the entries for the objects themselves must contain the header of the list of associated constraint checks. When constraints are known to be constant, it is assumed that those constants have entries in the symbol table which will act as headers of the lists.

In order to limit the number of comparisons required to determine whether a new test is redundant, it is further assumed that tests against the same constraint are linked together, and that each such constraint has a separate entry in the symbol table which acts as header of the list. An alternate mechanism is to add another type of list item which acts as header of the list for tests against the particular constraint.

The symbol table is also used as the anchor for information required by the induction variable reduction algorithm. A control block is attached to any scalar variable that may be processed as an integer by the generated code. When a new loop is entered, a new control block is stacked onto the symbol table entry for the variable, if the variable is referenced in the loop. The control block has the form:

```
type induction_variable_info is record
    depth : Integer;        -- nesting depth in which variable is referenced
    iv_type : (loop_parameter, basic, auxiliary, invalid);
    multiplier: Integer;    -- primarily for auxiliary induction variable
    increment: Integer;
    biv: access induction_variable_info;    -- pointer to basic induction
                            -- variable if this variable is an
                            -- auxiliary induction variable
    biv_increment: Integer;    -- for auxiliary induction variables
    family: symtab_entry;   -- list of aux induction variables if this
                            -- variable is a basic induction variable
    previous_info: access induction_variable_info; -- pointer to info for
                            -- this variable in some outer loop.
end record;
```

6.4 Syntax Tree

In addition to the constraint checks defined above, additional fields need to be added to the nodes in the syntax tree. Nodes representing variables need an entry for the variable's current

store number. Statement nodes need a pointer for the list of tests associated with the statement. Nodes for loop statements need a header for the list of variables redefined in the loop.

6.5 Miscellaneous Data Structures

```
level: Positive;          --indicates nesting level of linear blocks
depth: Positive;          --indicates nesting level of loops

type loop_entry is record
    exit_level: Positive;
    store#: store_num;    --last valid store number prior to entry into loop
    label: symtab;        --loop identifier
end record;

loop_stack: array(1..max_depth) of loop_entry;
level_stack: array(1..max_level) of Positive; --last valid store number
                          --prior to entry into new level
```

7 Algorithms

7.1 Driver and Test Elimination Routine

```
procedure main_routine is
  store#: Positive := 1;
  L: value;
begin
  --Initialization
  for every scalar variable V in the symbol table loop
    if V has an initial value then
      L.level := constant;
      L.actual := V'initial_value;
      new_store#(V,L);
    else
      --Indicate that the variable is not initialized by giving the
      --variable a negative store#. There is no mapping for negative
      --store#'s.
        V.store# := -1;
    end if;
  end loop;

  --Set level of outermost linear block in procedure
    level := 1;
  --Set (loop) depth of outermost block
    depth := 1;
```

```
  -- Stack store# for outermost block
     push(store#);
  -- Process procedure beginning with first executable statement, S, in
  -- the outermost linear block.
     process_block(S,gen,kill,exit_flag);
end main_routine;


procedure process_block(S: ast_node; gen, kill: in out list_of_items;
        block_exited: out Boolean) is
   nested_gen,nested_kill: list_of_items; -- used for nested linear blocks
   stmt_gen, stmt_kill: list_of_items; -- used for conditional statements
   stmt: ast_node;
   exited, stmt_exited : Boolean;
begin
   stmt := S;
   gen := null;
   kill := null;
   block_exited := false;
   loop
      exit when reach end of linear block;
      case stmt'type is
         when simple_statement (e.g. assignment, procedure call) ⇒
            inspect(stmt,candidate_list); -- build list of checks for statement
            for every test T in candidate_list loop
               if not test_live(T) then
                  -- The constraint has not been checked by a currently valid
                  -- test. Create the test and insert it into the appropriate
                  -- place in the tree.
                     Create_test(T,gen,node);
               end if;
            end loop ;

            for every variable V defined in the statement loop
               Compute the new lattice value L of V;
               Update_store#(V,L); -- give new store# to V
               if store#_table(V.store#).prev ≤ store#_stack(top) then
                  -- Variable has not been changed previously in this block.
                  -- Add entry to gen list for processing after block has been
                  -- exited.
                     Append store# to gen;
               end if;
               Kill_all_tests(V,kill,level); -- kill all tests on uselist(V)        (A.1)
            end loop ;
```

```
-- End when simple_statement

when conditional_statement ⇒
  -- Because of short-circuit operators, it is difficult to keep
  -- track of which tests are available to later code. Call a
  -- separate routine to determine which tests are available. Also,
  -- create pseudo-tests for constraint tests explicitly coded by
  -- the programmer.
    process_boolean(condition,tree_position);

  -- Initialize exit flags in the event that there is an exit of
  -- some enclosing loop from within the conditional statement.
    exited := false;
    stmt_exited := false;
    stmt_exit_depth := loop_stack(depth).exit_level;

  -- Process linear block associated with condition. Stack store#
  -- in order to recognize any assignments within the linear block.
  -- Process_block returns lists of the tests generated and killed
  -- within the conditional so that the state may be reset for
  -- other linear blocks within the conditional statement.
    stack(store#);
    process_block(linear_block,nested_gen,nested_kill,exited);
    pop(store#);

  -- Kill all tests and definitions created in the linear block and
  -- live on exit from the block, so that other conditional blocks
  -- will not erroneously believe that a constraint has already
  -- been tested, or that a variable has a lower lattice value.
  -- But remember the changes to correctly set the program state
  -- upon completion of the statement.
    stmt_gen := null;
    stmt_kill := null;
    for every entry E in nested_gen loop
      if E is a test and then not E.kill then
        Add E to stmt_gen;
        E.kill := true;
      elsif E is a variable definition then
        if the variable exited the block with a level of CONSTANT
        or SIGN then
          Add an entry for that last store# to stmt_gen;
        else
          Add the associated variable to the stmt_kill list;
          Set join flag for variable in symbol table;
```

```
        end if;
      end if;
    end loop ;
--  Restore killed tests for use by other parts of the conditional
--  but add them to stmt_kill list so that they may be easily killed
--  after the entire conditional statement has been processed.
    for every test T in nested_kill loop
      Add T to stmt_kill;                                          (C.1)
      T.kill := false;
    end loop ;
--  Check whether linear block contained an exit from some
--  enclosing loop. If so, save that information to be processed
--  at the end of the conditional statement.
    if exited then                                                 (C.5)
      stmt_exited := true;
      stmt_exit_depth := min(loop_stack(depth).exit_level,
          stmt_exit_depth);
      exited := false;
    end if;
--  Process each remaining conditional component of the statement
    for each conditional part loop
      process_boolean(condition,tree_position);
      stack(store#);
      process_block(linear_block,nested_gen,nested_kill,exited);
      pop(store#);
      --  Delete all tests in stmt_gen for which an identical
      --  condition was not tested in the linear block just processed.
      --  It is not required that the values of the component
      --  variables of the tested expressions are the same.
        for every test T in stmt_gen loop                          (C.4)
          if there is no live, congruent, test in nested_gen then
              Remove T from stmt_gen;
          end if;
        end loop ;
      --  Process the entries in the nested_gen list. Kill all
      --  tests created in the linear block. Also, compute the meet
      --  of lattice values for variables assigned in the linear block.
        for every entry E in nested_gen loop
          if E is a test then
            E.kill := true;
          else --  E is a variable definition
            if the join flag is on in the symbol table for the
            associated variable then --  the variable has already been
              null;                  --  killed.
```

```
        else
          Scan stmt_gen for an entry for the same variable;
          if an entry E2 is found in stmt_gen then
            meet := compute_meet(E,E2);
            if meet = bottom then
              Set the join flag in the symbol table entry for
                the associated variable;
              Add the variable to the stmt_kill list;
            else
              Create a new store# entry, E_NEW, with the
                result of the meet;
              Replace E2 in stmt_gen with E_NEW;
              E_NEW.VISITED := true; -- indicate valid for
                                     -- latest block.
            end if;
          else -- variable not changed in earlier conditionals.
            Look at last store# E0 for variable prior to
              conditional statement;
            if value for E0 is not bottom then
              meet := compute_meet(E,E0);
              if meet /= bottom then
                Create a new store# entry, E_NEW, with the
                  result of the meet;
                Add E_NEW to stmt_gen;
                E_NEW.VISITED := true;
              end if;
            end if;
          end if;
        end if;
      end if;
    end loop ;
  -- Check that all store# entries on stmt_gen are valid for
  -- the current conditional linear block.
    for every entry E in stmt_gen loop
      if E.visited = false then -- variable not changed in last
                                -- linear block.
        Let E0 be the last store# for the variable prior to the
          conditional statement;
        meet := compute_meet(E,E0);
        if meet /= bottom then
          Create a new store#, E_NEW, for the variable with
            meet and set backward reference to last store# for
            variable prior to conditional statement (symbol
            table entry for variable is NOT updated);
```

```
            Replace E2 in stmt_gen with E_NEW;
          else
            Remove E from stmt_gen;
            Set join flag in symbol table entry for variable;
          end if;
        else -- E visited, reset flag.
          E.visited := false;
        end if;
      end loop ;
    -- Restore external tests killed in linear block for use by
    -- other parts of the conditional.
      for every test T in nested_kill loop
        Add T to stmt_kill;                                        (C.2)
        T.kill := false;
      end loop ;

    -- Check whether the linear block contained an exit from some
    -- enclosing loop. If so, save that information to be processed
    -- at the end of the conditional statement.
      if exited then
        stmt_exited := true;                                       (C.6)
        stmt_exit_depth :=
          min(loop_stack(depth).exit_level,stmt_exit_depth);
        exited := false;
      end if;
  end loop ; -- for each successive conditional part

-- Summarize the results of the conditional statement
  -- Kill any external tests killed in any of the linear blocks.
  -- Also, set any variable which had a lattice value of bottom
  -- after any of the linear blocks to bottom for the
  -- conditional statement.
    for every entry E in stmt_kill loop
      case E'type is
        when test ⇒
          E.kill := true;                                          (C.3)
        when variable ⇒
          Create a new store# entry for the variable,
            with level set to bottom;
          Clear the join flag in the symbol table;
      end case;
    end loop ;

-- Update the store# for every variable changed in the
```

```
-- conditional statement, and exiting with a lattice level
-- higher than bottom.
  for every store# N in stmt_gen loop
    Store N in the symbol table entry for the
      associated variable;
  end loop ;

-- Make tests common to all parts available to the rest of the
-- program.
  for every test T in stmt_gen loop
    Create pseudo test PT identical to T except that the
      current store# for each referenced variable is used;
    Append PT to gen and to each of the appropriate
  uselists;
  end loop ;

-- Check if any of the linear blocks in the conditional
-- contained an exit from an enclosing loop. If so, process
-- the remainder of the linear block as if the conditional
-- statement were itself an exit statement.
  if stmt_exited then
    loop_stack(depth).exit_level := stmt_exit_depth;
    Process_exit(stmt'succ,gen,kill);                          (C.7)
    block_exited := true;
    return;
  end if;
-- end when conditional statement

  when loop ⇒
    -- If this loop is at depth = 1, call prepass to build a list of
    -- all variables defined in the loop, including all nested loops.
    -- The list, killed_variables, for each nested loop is
    -- attached to its loop node.
      if depth = 1 then
        prepass(loop_body,killed_variables);
      end if;
    -- Create new store#'s for each variable on the kill list to
    -- prevent external tests from being mistakenly used in the loop
      stack(store#);
      for every variable V in killed_variables loop
        update_store#(V,(bottom,0));
        kill_all_tests(V,kill,level+1);                        (L.1)
      end loop ;
      if for_loop(stmt) then
```

```
    -- Give new store# to loop parameter to keep dependent tests out
    -- of loop prologue. Will later look explicitly for tests
    -- dependent on the loop parameter or other induction variables.
      update_store#(loop_parameter,(bottom,0));
  end if;
-- Process boolean expressions for for and while loops
  if for_loop(stmt) or else while_loop(stmt) then
    process_boolean(condition,tree_position);
  end if;
-- Process loop body
  depth := depth + 1;
  level := level + 1;
  loop_stack(depth).exit_level := depth + 1;  -- indicate loop not
                                              -- exited
  process_block(loop_body,nested_gen,nested_kill,exited);          (L.4)
  level := level - 1;
  depth := depth - 1;
  pop(store#);
  if for_loop(stmt) then
    Process induction variables;
  end if;
-- Move loop independent tests to the loop preheader so that
-- they are executed only once. Set their levels to the level
-- of the current linear block. Also, check whether each test
-- is independent of the outer loop, if there is one.
-- Reset the prologue flag as appropriate.
  for every test T in nested_gen loop                              (L.2)
    if T.prolog then
      Move T to preheader list;
      T.level := level;
      if T.store# > level_stack(top) then
        T.prolog := false;
      end if;
    end if;
  end loop ;
  if 0_trip_loop(loop_condition) then
    -- Loop may not be executed. Kill all tests generated in the
    -- loop.
    for every test T in nested_gen loop
      T.kill := true;
    end loop ;
    -- The lattice level of each variable defined in
    -- the loop must be set to the meet of its level
    -- immediately prior to entering the loop and its
```

```
        --final level within the loop.
          for every definition in nested_gen loop
            if the final value for the variable associated with the
            store# is bottom then
              Reset backward pointer in store#_table to last store
                 prior to loop;
            else --Final value is constant or sign.
              Take meet of final value within loop and final value prior
                to loop (found by following the backward link
                in the store# table);
              Create a new store# entry for the result of the meet and
                save the new store# in the symbol table entry for the
                associated variable. Backward link the new store# to
                the last store# prior to the loop;
            end if;
          end loop ;
      else
        --1-trip loop. Move all live entries in nested_gen to gen.
          for every entry E in nested_gen loop
            if not E.kill then
              Move E to gen;
              if E.prolog then
                 Move E to loop prologue;
              end if;
            end if;
          end loop ;
      end if;
    --Check if the loop contained an exit from an enclosing loop.
    --If so, tests and definitions in the remainder of the current
    --linear block cannot be made available outside.
      loop_stack(depth).exit_level :=                              (L.5)
        min(loop_stack(depth).exit_level,
         loop_stack(depth+1).exit_level);
      if loop_stack(depth).exit_level ≤ depth then
        Process_exit(stmt'succ,gen,kill);                          (L.3)
        block_exited := true;
        --Process_exit processes all remaining statements in the
        --linear block. Return to caller.
        return;
      end if;
  --end when loop

  when exit_statement ⇒
  --No tests or definitions following the exit statement can be
```

```
        -- considered available outside the current linear block.
        -- Process the remainder of the block separately so that all
        -- tests that follow can be easily killed. Also, all variables
        -- defined after the exit will be given new store numbers for the
        -- meet of their values before and after the exit.
            process_boolean(when_condition,tree_position);
            process_exit(after_exit,gen,kill);                    (E.1)
            block_exited := true;                                 (E.2)

          -- Return to caller; All other statements in current linear
          -- block have been processed.
            return;
      -- end when exit_statement

    end case;

  -- Go on to the next statement
    stmt := ast_node'succ(stmt);
 end loop ;
 return;
end process_block;
```

7.2 Induction Variable Analysis

Processing for induction variable analysis occurs in several separate areas of the total algorithm: during the prepass, upon entry into a **for** loop, while processing statements during the loop, and upon exit from the loop. Each phase will be discussed here rather than incorporated with the test analysis algorithm to (hopefully) make the ideas clearer.

7.2.1 Prepass Processing

We earlier said that the purpose of the prepass is to determine which variables are killed in a loop. We may also determine the type of each killed variable, as well as how the variable was killed, to screen out variables which could not possibly be induction variables and, thereby, eliminate useless processing.

7.2.2 FOR Statement Processing

Prior to processing the body of a **for** loop, the loop parameter is identified as a basic induction variable, i.e., of the form $i := i + c$; where c = 1 or −1. Then as each assignment statement within the body of the loop is processed, the following actions are taken:

When processing a **for** statement:
Add pseudo constraint checks for the loop parameter.
Mark the loop parameter i as a basic induction variable.
Add the loop parameter to the list of basic induction variables with

template $(i, i, 1, 1)$ or $(i, i, 1, -1)$, depending on the direction of the loop.

7.2.3 Loop Body Processing

When processing each statement in the loop, in addition to determining the constraint checks required, we determine whether the statement creates any induction variables, or invalidates any variables as induction variables.

Process assignment statements in the **for** loop as follows:

if the statement is of the form $i := i + c;$, c constant, **then**
  if i was already marked as a basic induction variable with
  formula $(i, i, 1, d)$ **then**
    if sign(c) = sign(d) **then** *--|i| is increasing through the loop*
      Update current formula to $(i, i, 1, c + d)$.
      Update current formula for every variable in family(i):
        if $j := a * i + b$ and j has current formula of (j, i, a, b) **then**
          Update formula to $(j, i, a, b + a * d)$;
        end if;
    else *--|i| is not increasing monotonically in the loop*
      Kill i as an induction variable—we no longer know its
        direction;
    end if;
  elsif i was marked as an auxiliary induction variable with
  current formula (i, k, l, m), i.e., $i = l * k + m$ and k is a basic induction
  variable **then**
    if sign(m) = sign(d) **then**
      Update current formula of i to $(i, k, l, m + d)$;
    else *--|i| is not increasing monotonically in the loop*
      Kill i as an induction variable—we no longer know its direction;
    end if;
  else
    Mark i as a basic induction variable with initial and current
      formulas $(i, i, 1, c)$;
    Initialize family(i) = empty;
  end if;
elsif statement is of the form $j = c * i + d$, c and d constant,
$|c| \neq 1$, **then**
  if j is already flagged as either a basic or auxiliary induction
  variable **then**
    --j vacillates within the loop, cannot determine maximum
    -- absolute change
    if j is marked as a basic induction variable **then**
      Invalidate all auxiliary variables in family(j);
    end if;
    Invalidate j as an induction variable;

```
    elsif i is a basic induction variable then
      Mark j as an auxiliary induction variable with formula (j, i, c, d)
        as both current and initial formulas of j;
      Add j to family of i.
    elsif i is an auxiliary induction variable in family(k), with
    formula (i, k, r, s) then
      Mark j as an auxiliary induction variable with
        formula (j, k, c * r, c * d + d);
      Add j to the family of k;
    end if;
  else
    Set symbol table entry to indicate that left hand side cannot be
      an induction variable;
  end if;
```

When creating a constraint check, determine if it is a candidate for replacement:

```
for every test T created for statement S loop
  if the expression tested is of the form a * i + b, where i is an
  induction variable, a and b are loop constants, and sign(a) is known
  then
    Create an entry on candidate list. Include in the list entry:
      Ast pointer to constraint check created
      Symbol table address of induction variable
      Current formula for induction variable
  end if;
end loop ;
```

7.2.4 End of Loop Processing

At the end of the loop, all entries on the candidate list are examined. If the referenced induction variable was not invalidated after the candidate list item was created, then an equivalent test is created in the loop prologue and the original test is discarded.

```
if the loop does not contain an exit statement or subroutine call then
  for every entry E in the candidate list loop
    if the referenced induction variable i is still valid then
      Create a test based on the tables in Appendix A;
      Delete the original test, pointed to by E;
    end if;
  end loop ;
end if;
```

Finally, the list of variables modified in the loop is inspected. If any of the variables were invalidated as induction variables, then they are invalidated in the containing loops as well. If

a variable is a valid induction variable, and the loop is a 1-trip loop with constant bounds, then the variable is marked as an induction variable in the containing loop.

```
if the loop is a nested 1-trip loop with constant range then
  for every variable defined in the loop loop
    if the variable i is a valid basic induction variable in the
    containing loop with final form i := i + c;, c constant, then
      Update the definition of i in the containing loop as indicated
        above, treating the current loop as the definition:
          i := i + c * loop_parameter'length;
      for every auxiliary variable j in family(i) loop
        if j := a * i + b, a and b constant, was not invalidated as an
        induction variable in the containing loop then
          -- Update definition of j in containing loop, applying the same
          -- tests as applied above
          if i was of the form i := i + c; when j was defined, and was
          not incremented afterward then
            Update induction variable status of j in containing loop,
              treating current loop as the definition:
                j := a * (i + loop_parameter'length * c) + b;
          else -- i was modified after j was defined
            Update induction variable status of j in containing loop,
              treating current loop as the definition:
                j := a * (i + c0 + (loop_parameter - 1) * (c - c0)) + b; where
              i was of the form i := i + c0; when j was defined
          end if;
        end if;
        Delete induction variable record of j for current loop;
      end loop ;
    end if;
    Delete induction variable record of i for current loop;
  end loop ;
```

7.3 Process Boolean Expressions

This routine handles constraint checks for boolean expressions found in conditional statements and loops. Because of the short-circuit control forms, it is difficult to determine exactly which constraint checks generated by the boolean expression are available to the following code. As a result, a special routine is needed to determine which tests will always be executed.

```
procedure process_boolean(expression: ast_node; place: ast_node) is
    gen,candidate_list: list_of_items := null;
    position : ast_node;
begin
    Inspect(expression,candidate_list);
```

```
    for every test T in candidate_list loop
        if not test_live(T) then
            position := point in tree where test is to be inserted;
            create_test(T,gen,position);
        end if;
    end loop ;

    if expression tests template fields then
        create_pseudo_test(condition,position);
    end if;

    for every test T in gen loop
        if T is covered by a short circuit control form then
            T.kill := true;
        else
            T.kill := false;
        end if;
    end loop ;
end process_boolean;
```

7.4 Exit Processing

This routine handles the code following an **exit** statement. All following statements within the current linear block are processed separately. All tests that follow the **exit** statement are killed to prevent their use outside the linear block. Any test that is killed in the following code remains dead (unless resurrected as required when processing conditional statements.) Any variable defined in the following code receives a new store number to reflect the meet of its value upon entry into the "exit block" and upon exit from the linear block. Note that this is not accurate, for in reality the definitions following the **exit** statement may not reach the statements immediately following the current linear block. But it is safe. By propagating the **exit** statement outward, and continuing to take meets, we guarantee that all variables have a lattice value $\leq$ its true value upon exit from the loop exited by the statement.

```
procedure process_exit(stmt: ast_node; caller_gen,caller_kill: in out list_of_items)
is
    nested_gen,nested_kill : list_of_items;
begin
    stack(store#);
    process_block(stmt,nested_gen,nested_kill);
    pop(store#);
    -- Delete all tests created following the exit statement, and
    -- update the values associated with each variable modified
    -- after the exit.
    for every entry E in nested_gen loop
        case E'type is
```

```
            when test ⇒
                E.kill := true;
            when variable definition ⇒
                Create a new store# for the associated variable with
                    value equal to the meet of the last store# prior to
                    the exit and its final store# upon return from process_block;
                Add store# entry to gen if it has not been changed
                    previously in the block;
        end case;
    end loop ;

    -- Add all external tests killed in code following exit to kill list
    -- for caller.
    caller_kill := caller_kill & nested_kill;

    exit_flag := true;
    loop_exited(depth) := min(loop_stack(depth).exit_level,exit_level);          (E.3)

    -- Return to caller; All other statements in the current linear
    -- block have been processed.
    return;
end process_exit;
```

7.5 Test Check Routine

This routine is called whenever one of the statement processors exposes a constraint check that requires examination of a template field. Its purpose is to determine whether the test needs to be inserted into the code, or whether it is testing a condition already validated by an existing test. Input is a record representing the test, including the type of test, the expression being tested, the maximum store number for any variable in the expression, and the template field against which the expression is being validated.

```
function test_live(CC: constraint_check) return Boolean is
begin
  for each test T in CC.template_field.uselist loop
    if not T.kill and not T.pseudo and T.store# = CC.store#
    and then congruent_expression(T.expression,CC.expression) then
      return true;
    else
      null;
    end if;
  end loop ;
  return false;
end test_live;
```

7.6 Update Store Number

This routine updates the store number for a variable to reflect the changes caused by an assignment, or the side effects of a procedure call. It creates a new entry for the store number table, including a pointer to the current store number entry for the variable, and updates the symbol table entry for the variable. Input is the address of the symbol table entry for the variable, and a lattice record containing the level and value to be associated with the store number.

```
procedure update_store#(V: symtab; L: value; gen: list_of_items) is
begin
    store_num_table(store#).V := V;
    store_num_table(store#).L := L;
    store_num_table(store#).prev := V.store#;
    V.store# := store#;
    store# := store# + 1;
end update_store#;
```

7.7 Test Creation Routine

This routine is called to insert a new test into the parse tree. Input to the routine is a test record, the gen list for the current linear block, and the position of the tree at which the test is to be inserted. In addition to creating the test, this routine adds the test to the uselists for the associated template field, and for every variable referenced in the expression being tested.

```
procedure create_test(T: constraint_check; gen: list_of_items;
stmt_node: ast_node) is
begin
    Build subtree for test;
    Insert subtree at stmt_node;
    Append an entry for the test on gen;
    Append an entry for the test on T.field.uselist;
    for every variable V referenced in T.expression loop
        Append an entry for the test on V.uselist;
        T.store# := max(V.store#,T.store#);
    end loop ;
    T.level := level;
    T.kill := false;
    T.pseudo := false;
    if T.store# > level_stack(top) then
        T.prolog := false;
    else
        T.prolog := true;
    end if;
end create_test;
```

7.8 Kill All Tests

This routine kills all tests on a variable's uselist. If the test was created outside the current linear block, then the killed test is appended to the kill list for processing after the linear block is exited.

```
procedure kill_all_tests(V: symtab; kill: list_of_items; level:Positive) is
begin
    for every test T in V.uselist loop
        if not T.kill then
            T.kill := true;
            if T.level ≠ level then
                -- Test was created in enclosing linear block
                Append T to kill;
            end if;
        end if;
    end loop ;
end kill_all_tests;
```

8 Proof of Correctness

Theorem 8.1 *Test Elimination via Available Expression Analysis is Safe.*

Proof: The algorithm eliminates a test at a statement S only when that the test was already executed at some point P which dominates S, and if none of the arguments referenced in the test have been modified between P and S. More precisely, to show that the algorithm is safe, we will show that a test

$$t = operation(expression, constraint)$$

is in $in(S)$ for some statement S if and only if

1. t was created at some point which dominates S.
2. t has not been invalidated prior to S through some redefinition of any variable referenced by t.

Case 8.1 *Loops and Conditional Statements Contain Simple Statements Only; Loops Contain No* **exit** *Statements*

We first consider the case of a program containing no **exit** statements. By assumption then, the program is made up of simple, conditional, and loop statements only. We prove that the algorithm is safe for such a program by induction on n = the number of loop or conditional statements in the program.

Subcase 8.1.1 $n = 0$

We assume that there are no conditional or loop statements in the program. Then each statement S is dominated by all statements appearing earlier in the program. Therefore, any test $t \in in(S)$ must have been created at a point which dominates S. Furthermore, t must still be valid at S. For suppose that t was initially created at S_0. Then $t \in in(S) \Rightarrow t \in out(S_i)$ for S_0 and all its successors up to S_n, the immediate predecessor of $S, \Rightarrow t \notin kill(S_i), i = 0, \ldots, n$. t would no longer be valid at S only if *expression* or *constraint* was modified prior to S. But should that have occurred, t would have been killed by the call to *kill_all_tests* at (A.1) on page 41.

Subcase 8.1.2 $n = 1$

The program contains only a conditional or a loop statement. First assume that the program contains a single conditional statement S_c preceding S someplace in the program.

Single Conditional Statement in Program. Let $t \in in(S)$. If t was created after S_c, then this case reduces to the $n = 0$ case, and the algorithm is safe. So suppose that $t \in in(S_c)$ and that some variable referenced by t was redefined in one or more conditional blocks of S_c. Then t is saved on the *stmt_kill* list at either (C.1) on page 43 or (C.2) on page 45 and killed at (C.3) on page 45 $\Rightarrow t \notin out(S_c) \Rightarrow t \notin in(S)$, contradicting our initial assumption. Therefore, $t \in in(S) \Rightarrow t$ was not killed in S_c. Finally, suppose $t \in in(S)$ and t was created in some conditional block of S_c. Then $t \in gen(C_c)$ for every conditional block C_c in S_c. For if not, then it would have been killed at (C.4) on page 43. Since all remaining statements between S_c and S are simple $\Rightarrow$ every path to S must have the test t on it.

Single Loop in Program. Now suppose instead that S was preceded by a loop, S_L, and $t \in in(S_L)$. If any operand of t is redefined in S_L then t is killed at (L.1) on page 46 $\Rightarrow t \notin out(S_L) \Rightarrow t \notin in(S)$. Therefore, $t \in in(S) \Rightarrow t$ was not killed in S_L. Finally suppose $t \in gen(S_L) - in(S_L)$. S_L cannot be a 0-trip loop, for all tests generated in a 0-trip loop are killed at (L.2) on page 47 $\Rightarrow S_L$ must be executed at least once (recall that there are no conditional statements in the program) $\Rightarrow$ every statement in the loop dominates $S \Rightarrow t \in in(S) \Rightarrow t$ has already been executed prior to S and none of its arguments have been modified.

Subcase 8.1.3 $n = k$

Suppose that the algorithm is safe for programs with $k - 1$ conditional and/or loop statements in series. We show that the algorithm is safe for a program with k such statements. Again, let $t \in in(S)$. Let S_{CL} be the last loop or conditional statement prior to S. Then the sequence of statements prior to S_{CL} contains $k - 1$ loop or conditional statements and the sequence of statements following S_{CL}, prior to S, contains only simple statements. By the proof for the $n = 1$ case, $t \in in(S) \Rightarrow t \in in(S_{CL})$ and was not killed between S_{CL} and S, or either S_{CL} is a conditional statement and t was generated in every conditional block of S_{CL}, or S_{CL} is a 1-trip loop. If $t \in in(S_{CL})$, then, by the induction hypothesis, t was created at a point which dominates S_{CL} and was not killed prior to S_{CL}. In either event, $t \in in(S) \Rightarrow t$ dominates S and is not killed prior to S.

Case 8.2 *Programs with nested loops and/or conditional statements*

So far we have stipulated that the conditional blocks and loop bodies were composed only of simple statements. We now employ induction to prove that the algorithm is safe for arbitrary nestings of loop and conditional statements. The above proof proves the case for programs at nesting level 1. Assume now that the hypothesis is true for programs with nests of up to $k-1$ conditionals or loops, and consider a program at nesting level k (i.e., the program has one or more conditional or loop statements with a linear block at nesting level $k-1$). Let S be one of those statements and let lb be one of those blocks at nesting level $k-1$. By the induction hypothesis, every test $t \in gen(lb)$ must have the properties that it is executed on every path through lb and that it is not invalidated prior to reaching the end of lb. Then by the proof above, $t \in out(S)$ if and only if t is executed on every path through S and not invalidated prior to the end of S.

Case 8.3 *Programs with* **exit** *statements*

Now let us consider the case for **exit** statements. We will prove that if t is a test which could be bypassed by some preceding **exit** statement in the body of a loop L, then $t \notin out(L) \Rightarrow t \notin in(S)$ for any statement S which follows the loop.

Let L be a linear block made up of one or more **exit** statements and 0 or more simple statements. Each **exit** in L causes *Process_exit* to be called at (E.1) on page 49 to kill all tests generated in L at any point following that **exit**. *Process_exit* also maintains the nesting depth of the outermost loop exited by any of the **exit** statements processed. Then $t \in gen(L) \iff t$ is generated prior to the first **exit** statement in L. *Process_block* also sets the *Block_exited* parameter to **true** at (E.2) on page 49 to notify the caller that control may leave L prior to its physical end.

Now suppose that L' is the linear block immediately containing L. When *Process_block* returns control to its caller in L', *Block_exited* is inspected. If L is the body of a 1-trip loop in L', *Process_block* checks if some outer loop is exited, and if so, calls *Process_exit* at (L.3) on page 48 to process the remainder of L'. If, instead, L is a conditional block of a conditional statement within L', the status of *Block_exited* is saved at (C.5) on page 43 or (C.6) on page 45 until all of the statement's conditional blocks have been processed. Then, after completing the processing for the conditional statement, *Process_exit* is called at (C.7) on page 46 to handle the remainder of L' and ensure that all tests generated within L', and after L, are killed before control returns to the linear block containing L'. Therefore, regardless of the type of statement in L' containing L, no test t created in L' at a point following L, will appear in $gen(L') \Rightarrow t \notin out(L')$. This effect of the initial **exit** statement is propagated to the caller of *Process_block* at (L.4) on page 47 at the outermost loop exited by any **exit** statement within it. Any **exit** statement, found after the first, has only the effect of extending the propagation of the first one. This is guaranteed at statements (L.5) on page 48 and (E.3) on page 54 which maintain the outermost depth of any **exit** statement found.

Summarizing then, because no test t created after the first **exit** statement in any linear block L can appear in $out(L)$, it can never appear in $in(S)$ for any statement S following the linear block containing the **exit** and so can never be eliminated at any point outside the linear block. Because the **exit** statement is propagated, no test generated within the outermost loop

O at any point following the first **exit** within the body of O can appear in $out(O) \Rightarrow$ that no such test is in $in(S)$ for any statement S following the loop in the program $\Rightarrow$ it will not be considered redundant if required at a later point $\Rightarrow$ that the algorithm is safe.

A Replacement Tests for Induction Variables

We assume that a loop contains a test of the form

$$le(a * i + b, c); \tag{1}$$

for some induction variable i, and a and b satisfying the constraints mentioned in Section 2.3. As we discussed earlier, the type of replacement test inserted into the loop preheader is determined by whether the induction variable i is: (1) a loop parameter, (2) a basic induction variable, or (3) an auxiliary induction variable. The tables that follow present the replacement tests. Replacement tests for ge and eq are analogous.

A.1 Loop Parameters

If i is a loop parameter, then the replacement test is a function of its slope, i.e., increasing or decreasing, and the sign of the multiplier a in the test. The replacement tests are given in Table 1 below.

i	$sign(a)$	Replacement Test
increasing	+	$le(a * i\text{'}\texttt{Last} + b, *)$
increasing	−	$le(a * i\text{'}\texttt{First} + b, *)$
decreasing	+	$le(a * i\text{'}\texttt{First} + b, *)$
decreasing	−	$le(a * i\text{'}\texttt{Last} + b, *)$

Table 1: Replacement Tests for Loop Parameter i

A.2 Basic Induction Variables

Here, i is assumed to be a basic induction variable defined by an assignment of the form

$$i := i + c;$$

The form of the replacement test is a function of the sign of the increment c and the sign of the multiplier a in the test being replaced. The replacements tests for i in this situation are presented in Table 2. In the table, lp is the loop parameter of the loop containing the original test ((1) above), and $lp\text{'}\texttt{Length}$ is the number of values in the range of lp.

$sign(c)$	$sign(a)$	Replacement Test
+	+	$le\Big(a * (i + c * lp\text{'}\texttt{Length}) + b, *\Big)$
+	−	$le\Big(a * (i + c) + b, *\Big)$
−	+	same as $(+, -)$
−	−	same as $(+, +)$

Table 2: Replacement Tests for Basic Induction Variable $i := i + c;$

A.3 Auxiliary Induction Variables

In this case, i is an auxiliary induction variable defined by an assignment of the form

$$i := \alpha * j + \beta;$$

where j is a basic induction variable defined by the assignment

$$j := j + c;$$

The replacement test is a function of (1) the sign of the multiplier α, (2) the sign of c—the increment of the basic induction variable j, and (3) the sign of a—i's multiplier in the test being replaced. The replacement tests are listed in Table 3 below. As in Table 2, lp is the loop parameter of the loop containing i and j, and lp'`Length` is the number of values in the range of lp.

$sign(c)$	$sign(\alpha)$	$sign(a)$	Replacement Test
+	+	+	$lc\Big(a * \Big(\alpha * j + \beta + c * (lp\text{'}\texttt{Length}) * \alpha\Big) + b, *\Big)$
+	+	−	$le\Big(a * (\alpha * j + \beta + c * \alpha) + b, *\Big)$
+	−	+	same as $(+, +, -)$
+	−	−	same as $(+, +, +)$
−	+	+	same as $(+, +, -)$
−	+	−	same as $(+, +, +)$
−	−	+	same as $(+, +, +)$
−	−	−	same as $(+, +, -)$

Table 3: Replacement Tests for Auxiliary Induction Variable $i := \alpha * j + \beta;$

References

[AK87] Randy Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[AKPW83] John R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *Conference Record of the Tenth ACM Symposium on Principles of Programming Languages*, pages 177–189, 1983.

[Bre87] Ron Brender. Ada Language Maintenance Committee Meeting Notes. May 1987. Pages AI-00315/19–24.

[CS70] John Cocke and J. T. Schwartz. *Programming Languages and Their Compilers: Preliminary Notes, Second Revised Version.* Courant Institute of Mathematical Sciences, New York University, April 1970.

[DH87] Robert B. K. Dewar and Paul Hilfinger. Ada Language Maintenance Committee Meeting Notes. May 1987. Pages AI-00315/27–29.

[DoD83] United States Department of Defense. *Reference Manual for the ADA Programming Language, ANSI/MIL-STD-1815A-1983.* 1983.

[Don81] J. J. Dongarra. Some LINPACK timings on the Cray-1. In Robert H. Kuhn and David A. Padua, editors, *Tutorial on Parallel Processing*, pages 363–380, IEEE Computer Society Press, 1981.

[Ges73] Charles Matthew Geschke. *Global Program Optimization.* PhD thesis, Carnegie-Mellon University, 1973.

[IBFW86] Jean D. Ichbiah, John G. P. Barnes, Robert J. Firth, and Mike Woodger. *Rationale for the Design of the Ada Programming Language.* U. S. Government, 1986.

[Kil73] Gary A. Kildall. A unified approach to global program optimization. In *Conference Record of the Third ACM Symposium on Principles of Programming Languages*, pages 194–206, 1973.

[KKLW80] D. J. Kuck, R. H. Kuhn, B. Leasure, and M. Wolfe. The structure of an advanced vectorizer for pipelined processors. In *Proceedings of the Fourth International Computer Software and Applications Conference (COMPSAC '80)*, pages 709–715, October 1980.

[MCM82] Victoria Markstein, John Cocke, and Peter Markstein. Optimization of range checking. *SIGPLAN Notices*, 17(6):114–119, June 1982. (Proceedings of the SIGPLAN '82 Symposium on Compiler Construction).

[Pow84] Michael L. Powell. A portable optimizing compiler for Modula-2. *SIGPLAN Notices*, 19(6):310–318, June 1984. (Proceedings of the SIGPLAN '84 Symposium on Compiler Construction).

[Shu87] Norman Victor Shulman. *The Semantics of Shared Variables in Parallel Programming Languages.* PhD thesis, New York University, 1987.

[SI77] Norihisa Suzuki and Kiyoshi Ishihata. Implementation of an array bound checker. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 132–143, 1977.

[Wel78] J. Welsh. Economic range checks in Pascal. *Software—Practice and Experience*, 8:85–97, 1978.

[WJW*75] William Wulf, Richard K. Johnsson, Charles B. Weinstock, Steven O. Hobbs, and Charles M. Geschke. *The Design of an Optimizing Compiler.* American Elsevier, 1975.